OPEN ACCESS

University of Dundee

University of Dundee

Coalgebraic logic programming

Komendantskaya, Ekaterina; Power, John; Schmidt, Martin

Published in:
Journal of Logic and Computation

DOI:
10.1093/logcom/exu026

Publication date:
2016

Licence:
CC BY

Document Version
Publisher's PDF, also known as Version of record

Link to publication in Discovery Research Portal

Citation for published version (APA):
Komendantskaya, E., Power, J., & Schmidt, M. (2016). Coalgebraic logic programming: from semantics to implementation. *Journal of Logic and Computation*, *26*(2), 745-783. https://doi.org/10.1093/logcom/exu026

General rights
Copyright and moral rights for the publications made accessible in Discovery Research Portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from Discovery Research Portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain.
• You may freely distribute the URL identifying the publication in the public portal.

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 05. May. 2023

Coalgebraic logic programming: from Semantics to Implementation

EKATERINA KOMENDANTSKAYA, *School of Computing, University of Dundee, Dundee DD1 4HN, UK.*

JOHN POWER, *Department of Computer Science, University of Bath, Bath BA27AY, UK.*

MARTIN SCHMIDT, *Institute of Cognitive Science, University of Osnabrück, Osnabrück 49069, Germany.*

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

Abstract

Coinductive definitions, such as that of an infinite stream, may often be described by elegant logic programs, but ones for which SLD-refutation is of no value as SLD-derivations fall into infinite loops. Such definitions give rise to questions of lazy corecursive derivations and parallelism, as execution of such logic programs can have both recursive and corecursive features at once. Observational and coalgebraic semantics have been used to study them abstractly. The programming developments have often occurred separately and have usually been implementation-led. Here, we give a coherent semantics-led account of the issues, starting with abstract category theoretic semantics, developing coalgebra to characterize naturally arising trees and proceeding towards implementation of a new dialect, CoALP, of logic programming, characterised by guarded lazy corecursion and parallelism.

Keywords: Logic programming, coalgebra, observational semantics, corecursion, coinduction, parallelism.

1 Introduction

The central algorithm of logic programming is *SLD*-resolution [33, 35, 46]. It is primarily used to obtain *SLD*-refutations; it is usually given least fixed point semantics; and it is typically implemented sequentially [35, 46].

All three of these traditions have been challenged over the years, for related reasons. For example, infinite streams of bits can be described naturally in terms of a logic program `Stream`:

```
                    bit(0) ←
                    bit(1) ←
stream(scons(x,y)) ← bit(x),stream(y)
```

SLD-resolution is of value here, but *SLD*-refutations are not, and that is standard for coinductively defined structures [18, 23, 35, 37, 45]. Consequently, least fixed point semantics, which is based on finiteness of derivations, is unhelpful. `Stream` can be given greatest fixed point semantics [35], but greatest fixed point semantics is incomplete in general, failing for some infinite derivations. `Stream` can alternatively be given coalgebraic semantics [6, 9] or observational semantics [9, 14]. Coalgebraic semantics is, in general, well-suited for describing parallel processes [22, 43].

In this article, we propose a single coherent, conceptual semantics-led framework for this, developing and extending three conference papers [27, 29, 30]. We start from the theoretical, with an

Vol. 26 No. 2, © The Author, 2014. Published by Oxford University Press.
This is an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/), which permits unrestricted reuse, distribution, and reproduction in any medium, provided the original work is properly cited.
Published online May 28, 2014 doi:10.1093/logcom/exu026

abstract category theoretic semantics for logic programming, and we proceed to the applied, ultimately proposing a new dialect, CoALP, of logic programming based on our abstract development. We do not change the definition of a logic program; we rather change the analysis of it. `Stream` is a leading and running example for us.

In more detail, a first-order logic program consists of a finite set of clauses of the form

$$A \leftarrow A_1, \ldots, A_n$$

where A and the A_i's are atomic formulae, typically containing free variables, and with the A_i's mutually distinct. In the ground case, i.e. if there are no free variables, such a logic program can be identified with a function $p : At \longrightarrow P_f(P_f(At))$, where At is the set of atomic formulae and p sends an atomic formula A to the set of sets of atomic formulae in each antecedent of each clause for which A is the head [6, 9, 22]. Such a function is called a coalgebra for the endofunctor $P_f P_f$ on the category *Set*. Letting $C(P_f P_f)$ denote the cofree comonad on $P_f P_f$, given a ground logic program qua $P_f P_f$-coalgebra, we characterize and-or parallel derivation trees [17, 19, 41] in terms of the $C(P_f P_f)$-coalgebra structure corresponding to p, see Section 3. And-or parallel derivation trees subsume *SLD*-trees and support parallel implementation and the *Theory of Observables* [9, 14].

The extension from ground logic programs to first-order programs is subtle, requiring new abstract category theory. Nevertheless, it remains in the spirit of the situation for ground logic programs. Our characterization of and-or parallel derivation trees does not extend from ground to arbitrary logic programs exactly, but it fails in particularly interesting ways: the relationship between and-or trees and ours is at the heart of the study. Indeed, the analysis of trees is fundamental to us. We end our abstract development by proving soundness, completeness, correctness and full abstraction results for coalgebraic semantics in Section 3.

Proceeding from the abstract to the applied, two aspects of logic programming that are both desirable and problematic in practice are corecursion and parallelism.

Many accounts of corecursion in logic programming, e.g. CoLP [18, 45], use explicit annotation of corecursive loops to terminate infinite derivations, see Section 4. In such accounts, inductive and coinductive predicates are labelled in order to make the distinction between admissible (in corecursion) and non-admissible (in recursion) infinite loops. But some predicates need to be treated as recursive or corecursive depending on the context, making annotation prior to program execution impossible. Example 4.3, extending `Stream`, illustrates this.

We propose an alternative approach to corecursion in logic programming: a new derivation algorithm based on the coinductive trees—structures directly inspired by our coalgebraic semantics. The resulting dialect CoALP is based on the same syntax of Horn-clause logic programming, but, in place of SLD-resolution, it features a new *coinductive derivation* algorithm. CoALP's lazy corecursive derivations and syntactic *guardedness* rules are similar to those implemented in lazy functional languages, cf. [5, 10, 15]. Unlike alternative approaches [18, 45], CoALP does not require explicit syntactic annotations of coinductive definitions. We discuss coinductive trees and derivations in Section 4. There, we prove soundness and completeness of CoALP relative to the coalgebraic semantics of Section 3.

Another distinguishing feature of logic programming languages is that they allow implicit parallel execution of programs. The three main types of parallelism used in implementations are *and-parallelism*, *or-parallelism* and their combination: [17, 19, 41]. However, many first-order algorithms are P-complete and hence inherently sequential [11, 24]. This especially concerns first-order unification and variable substitution in the presence of variable dependencies. Care is required here. For example, in `Stream`, the goal `stream(scons(x, scons(y,x)))`, if processed sequentially, leads to a failed derivation owing to ill-typing, whereas if proof search proceeds in a

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

parallel fashion, it may find substitutions for x, e.g. 0 and `scons(y,z)`, in distinct parallel branches of the derivation tree, but such a derivation is not sound, see Example 5.1.

Existing implementations [17, 19, 41] of parallel *SLD*-derivations require keeping records of previous substitutions and so involve additional data structures and algorithms that coordinate variable substitution in different branches of parallel derivation trees; which ultimately restricts parallelism. If such synchronization is omitted, parallel *SLD*-derivations may lead to unsound results as in `Stream` above. Again, this can be seen as explicit resource handling, where resources are variables, terms, and substitutions. In Kowalski's terms [33], *Logic Programming = Logic + Control*. This leads to the separation of issues of logic (unification and *SLD*-resolution) and control (underlying implementation tools) in most parallel logic programming implementations, as we explain in Section 5.

CoALP offers an alternative solution to this problem. The coinductive resolution of CoALP has an inherent ability to handle parallelism. Namely, coinductive trees with imposed guardedness conditions provide a natural formalism for parallel implementation of coinductive derivations. Parallelization of CoALP is sound by (guarded) program construction and the construction of coinductive trees. The main distinguishing features of parallelism in CoALP are implicit resource handling and convergence of the issues of logic and control: no explicit scheduling of parallel processes is needed, and parallelization is handled by the coinductive derivation algorithm. We explain this in Section 5.

Ultimately, in Section 6, we propose the first implementation of CoALP, available for download from [44]. Its main distinguishing features are guarded corecursion, parallelism and implicit handling of corecursive and parallel resources. In Section 7 we conclude and discuss future work.

2 SLD derivations and trees they generate

We recall the definitions surrounding the notion of *SLD*-derivation [35], and we consider various kinds of trees the notion generates.

2.1 Background definitions

DEFINITION 2.1
A *signature* Σ consists of a set of *function symbols* $f, g, \ldots$ each equipped with an *arity*. The arity of a function symbol is a natural number indicating the number of arguments it has. Nullary (0-ary) function symbols are called *constants*.

Given a countably infinite set *Var* of variables, denoted x, y, z, sometimes with indices $x_1, x_2, x_3, \ldots$, terms are defined as follows.

DEFINITION 2.2
The set $Ter(\Sigma)$ of *terms* over Σ is defined inductively:

- $x \in Ter(\Sigma)$ for every $x \in Var$.
- If f is an n-ary function symbol and $t_1, \ldots, t_n \in Ter(\Sigma)$, then $f(t_1, \ldots, t_n) \in Ter(\Sigma)$.

DEFINITION 2.3
A *substitution* is a function $\theta : Ter(\Sigma) \to Ter(\Sigma)$ which satisfies

$$\theta(f(t_1, \ldots, t_n)) = f(\theta(t_1), \ldots, \theta(t_n))$$

for every n-ary function symbol f.

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

An *alphabet* consists of a signature Σ, the set *Var*, and a set of *predicate symbols* $P_1, P_2, \ldots$, each assigned an arity. If P is a predicate symbol of arity n and $t_1, \ldots, t_n$ are terms, then $P(t_1, \ldots, t_n)$ is a *formula*, also called an *atomic formula* or an *atom*. The *first-order language* $\mathcal{L}$ given by an alphabet consists of the set of all formulae constructed from the symbols of the alphabet.

DEFINITION 2.4
Given a first-order language $\mathcal{L}$, a *logic program* consists of a finite set of *clauses* of the form $A \leftarrow A_1, \ldots, A_n$, where A is an atom and $A_1, \ldots A_n$ $(n \geq 0)$ are distinct atoms. The atom A is called the *head* of the clause, and $A_1, \ldots, A_n$ is called its *body*. Clauses with empty bodies are called *unit clauses*. A *goal* is given by $\leftarrow A_1, \ldots A_n$, where $A_1, \ldots A_n$ $(n \geq 0)$ are distinct atoms.

Logic programs of Definition 2.4 are also called Horn-clause logic programs [35].

EXAMPLE 2.5
Program `Stream` from Introduction defines infinite streams of binary bits. Its signature consists of two constants, `0` and `1`, and a binary function symbol `scons`. It involves two predicate symbols, `bit` and `stream`, and it has five atoms, arranged into three clauses, two of which are unit clauses. The body of the last clause contains two atoms.

EXAMPLE 2.6
`ListNat` denotes the logic program

```
          nat(0) ←
     nat(s(x)) ← nat(x)
          list(nil) ←
list(cons(x,y)) ← nat(x), list(y)
```

Operational semantics for logic programs is given by *SLD*-resolution, a goal-oriented proof-search procedure.

DEFINITION 2.7
Let S be a finite set of atoms. A substitution θ is called a *unifier* for S if, for any pair of atoms A_1 and A_2 in S, applying the substitution θ yields $A_1\theta = A_2\theta$. A unifier θ for S is called a *most general unifier* (mgu) for S if, for each unifier σ of S, there exists a substitution γ such that $\sigma = \theta\gamma$. If θ is an mgu for A_1 and A_2, moreover, $A_1\theta = A_2$, then θ is a *term-matcher*.

We assume that, given a goal $G = \leftarrow B_1, \ldots, B_n$, there is an algorithm that, given $B_1, \ldots, B_n$, outputs B_i, $i \in \{1, \ldots, n\}$. The resulting atom B_i is called the *selected atom*. Most PROLOG implementations use the algorithm that selects the left-most atom in the list $B_1, \ldots, B_n$ and proceeds inductively.

DEFINITION 2.8
Let a goal G be $\leftarrow A_1, \ldots, A_m, \ldots, A_k$ and a clause C be $A \leftarrow B_1, \ldots, B_q$. Then G' is *derived* from G and C using mgu θ if the following conditions hold:

- θ is an mgu of the selected atom A_m in G and A;
- G' is the goal $\leftarrow (A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta$.

A clause C_i^* is a *variant* of the clause C_i if $C_i^* = C_i\theta$, with θ being a variable renaming substitution such that variables in C_i^* do not appear in the derivation up to G_{i-1}. This process of renaming variables is called *standardizing the variables apart*; we assume it throughout the article without explicit mention.

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

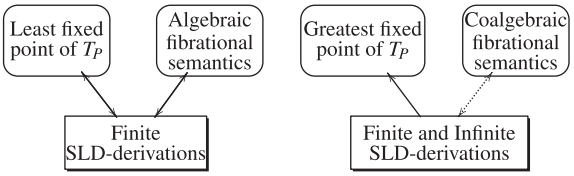

FIGURE 1. Alternative semantics for finite and infinite SLD-derivations. The arrows $\leftrightarrow$ show the semantics that are both sound and complete, and the arrow $\rightarrow$ indicates sound incomplete semantics. The dotted arrow indicates the sound and complete semantics we propose here.

DEFINITION 2.9
An *SLD-derivation* of $P \cup \{G\}$ consists of a sequence of goals $G = G_0, G_1, \ldots$ called *resolvents*, a sequence $C_1, C_2, \ldots$ of variants of program clauses of P, and a sequence $\theta_1, \theta_2, \ldots$ of mgu's such that each G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1}. An *SLD-refutation* of $P \cup \{G\}$ is a finite *SLD*-derivation of $P \cup \{G\}$ for which the last goal G_n is empty, denoted by $\Box$. If $G_n = \Box$, we say that the refutation has length n. The composite $\theta_1\theta_2, \ldots$ is called a *computed answer*.

Traditionally, logic programming has been modelled by *least fixed point* semantics [35]. Given a logic program P, one lets B_P (also called a *Herbrand base*) denote the set of atomic ground formulae generated by the syntax of P, and one defines $T_P(I)$ on 2^{B_P} by sending I to the set $\{A \in B_P : A \leftarrow A_1, ..., A_n$ is a ground instance of a clause in P with $\{A_1, ..., A_n\} \subseteq I\}$. The least fixed point of T_P is called the *least Herbrand model* of P and duly satisfies model-theoretic properties that justify that expression [35].

SLD-resolution is sound and complete with respect to least fixed point semantics [35]. The classical theorems of soundness and completeness of this operational semantics [12, 13, 35] show that every atom in the set computed by the least fixed point of T_P has a finite SLD-refutation, and vice versa. Alternatively, in [26, 28], we described an algebraic (fibrational) semantics for logic programming and proved soundness and completeness results for it with respect to SLD-resolution. Other forms of algebraic semantics for logic programming have been given in [2, 8]. See also Figure 1.

However, Programs like `Stream` induce infinite SLD-derivations and require a greatest fixed point semantics. The greatest fixed point semantics for SLD derivations yields soundness, but not completeness results.

EXAMPLE 2.10
The program `Stream` is characterized by the greatest fixed point of the T_P operator, which contains `stream(scons`$^{\omega}$`(X,Y))`; whereas no infinite term can be computed via SLD-resolution.

EXAMPLE 2.11
For the program $R(x) \leftarrow R(f(x))$, the greatest fixed point of the T_P operator contains $R(f^{\omega}(a))$, but no infinite term is computed by SLD-resolution.

There have been numerous attempts to resolve the mismatch between infinite derivations and greatest fixed point semantics [18, 23, 35, 37, 45]. Here, extending [29, 30], we give a uniform semantics of infinite SLD derivations for both finite and infinite objects, see Figure 1. Coalgebraic semantics has been used to model various aspects of programming [22, 39, 43], in particular, logic programming [6, 9]; here, we use it to remedy incompleteness for corecursion.

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

2.2 *Tree structures in analysis of derivations*

Coalgebraic Logic Programming (CoALP) we introduce in later sections uses a variety of tree-structures both for giving semantics to logic programming and for implementation of CoALP. Here, we briefly survey the kinds of trees traditionally used in logic programming.

For a given goal G, there may be several possible *SLD*-derivations as there may be several clauses with the same head. The definition of *SLD-tree* allows for this as follows.

DEFINITION 2.12
Let P be a logic program and G be a goal. An *SLD-tree* for $P \cup \{G\}$ is a possibly infinite tree T satisfying the following:

(1) the root node is G
(2) each node of the tree is a (possibly empty) goal
(3) if $\leftarrow A_1, \ldots, A_m, m > 0$ is a node in T, and it has n children, then there exists $A_k \in A_1, \ldots, A_m$ such that A_k is unifiable with exactly n distinct clauses $C_1 = A^1 \leftarrow B_1^1, \ldots, B_q^1, \ldots, C_n = A^n \leftarrow B_1^n, \ldots, B_r^n$ in P via mgu's $\theta_1, \ldots \theta_n$, and, for every $i \in \{1, \ldots n\}$, the ith child node is given by the goal

$$\leftarrow (A_1, \ldots, A_{k-1}, B_1^i, \ldots, B_q^i, A_{k+1}, \ldots, A_m)\theta_i$$

(4) nodes which are the empty clause have no children.

Each branch of an *SLD*-tree is an *SLD*-derivation of $P \cup \{G\}$. Branches corresponding to successful derivations are called *success branches*, branches corresponding to infinite derivations are called *infinite branches*, and branches corresponding to failed derivations are called *failure branches*. A distinctive feature of the SLD-trees is that they allow to exploit alternative choices of clauses in the proof-search; for this reason, they are also known as *or-trees*. See Figure 2.

In parallel logic programming [19], or-parallelism is exploited when more than one clause unifies with the goal. It is thus a way of efficiently searching for solutions to a goal, by exploring alternative solutions in parallel. It has been implemented in Aurora [36] and Muse [1], both of which have shown good speed-up results over a considerable range of applications.

Each *SLD*-derivation, or, equivalently, each branch of an *SLD*-tree, can be represented by a proof-tree, defined as follows.

DEFINITION 2.13
Let P be a logic program and $G = \leftarrow A$ be an atomic goal. A *proof-tree* for A is a possibly infinite tree T such that

- A is the root of T.
- Each node in T is an atom.
- For every node A' occurring in T, if A' has children $C_1, \ldots, C_m$, then there exists a clause $B \leftarrow B_1, \ldots, B_m$ in P such that B and A' are unifiable with mgu θ, and $B_1\theta = C_1, \ldots, B_m\theta = C_m$.

Proof-trees exploit the branching occurring when one constructs derivations for several atoms in a goal; and are also known as *and-trees*. In parallel logic programming, and-parallelism arises when more than one atom is present in the goal. That is, given a goal $G = \leftarrow B_1, \ldots B_n$, an and-parallel algorithm for *SLD*-resolution looks for *SLD*-derivations for each B_i simultaneously, subject to the condition that the atoms must not share variables. Such cases are known as independent and-parallelism. Independent and-parallelism has been successfully exploited in &-PROLOG [20].

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

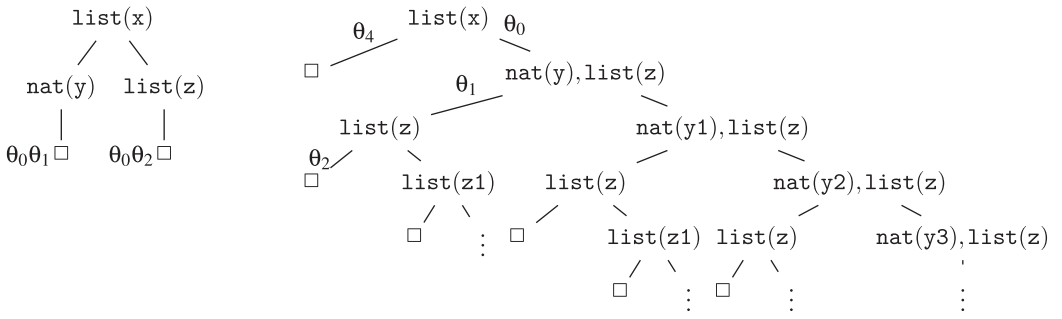

FIGURE 2. A proof tree and an *SLD*-tree for `ListNat` with the goal *list*(*x*). A possible computed answer is given by the composition of $\theta_0 = x/cons(y,z)$, $\theta_1 = y/0$, $\theta_2 = z/nil$; Another computed answer is $\theta_4 = x/nil$.

EXAMPLE 2.14
Figure 2 depicts a proof tree and an SLD-tree for the goal `list(x)` in `ListNat`.

EXAMPLE 2.15
`Stream` , i.e. Example 2.5, allows the following infinite *SLD*-derivation

$$\texttt{stream(x)} \xrightarrow{x/scons(y,z)} \texttt{bit(y)}\,,\texttt{stream(z)} \xrightarrow{y/0} \texttt{stream(z)} \rightarrow \ldots$$

containing an infinite repetition of `stream(x)` for various variables *x*. So `Stream` gives rise to infinite *SLD*-trees.

The and-trees, or-trees and their combination have been used in parallel implementations of logic programming, [17, 19, 41]. The main idea was that branches in the *SLD*-trees and proof-trees can be exploited in parallel. For certain cases of logic programs, such as ground logic programs or some fragments of DATALOG programs, one can do refutations for all the atoms in the goal in parallel [24, 32, 48]. But in general, *SLD*-resolution is P-complete, and hence inherently sequential [11].

The next definition formalizes the notion of and-or parallel trees [17, 19], but we restrict it to the ground cases, where such derivations are sound.

DEFINITION 2.16
Let *P* be a ground logic program and let $\leftarrow A$ be an atomic goal (possibly with variables). The *and-or parallel derivation tree* for *A* is the possibly infinite tree *T* satisfying the following properties.

- *A* is the root of *T*.
- Each node in *T* is either an and-node or an or-node.
- Each or-node is given by •.
- Each and-node is an atom.
- For every node A' occurring in *T*, if A' is unifiable with only one clause $B \leftarrow B_1, \ldots, B_n$ in *P* with mgu θ, then A' has *n* children given by and-nodes $B_1\theta, \ldots B_n\theta$.
- For every node A' occurring in *T*, if A' is unifiable with exactly $m > 1$ distinct clauses $C_1, \ldots, C_m$ in *P* via mgu's $\theta_1, \ldots, \theta_m$, then A' has exactly *m* children given by or-nodes, such that, for every $i \in \{1, \ldots, m\}$, if $C_i = B^i \leftarrow B^i_1, \ldots, B^i_n$, then the *i*th or-node has *n* children given by and-nodes $B^i_1\theta_i, \ldots, B^i_n\theta_i$.

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

Examples of and-or trees are given in Figures 3 and 9. In Section 5, we return to the questions of parallelism for CoALP.

3 Coalgebraic semantics

In this section, we develop the coalgebraic semantics of logic programming, starting from the coalgebraic calculus of infinite trees, through to the observational semantics of SLD-derivations.

3.1 A coalgebraic calculus of infinite trees

For the purposes of this article, a *tree* T consists of a set T_n for each natural number n, together with a function $\delta_n : T_{n+1} \longrightarrow T_n$, yielding

$$\ldots T_{n+1} \longrightarrow T_n \longrightarrow \ldots \longrightarrow T_1 \longrightarrow T_0 = 1$$

An element of T_n is called a *node* of T at height n. The unique element of T_0 is the *root* of the tree; for any $x \in T_{n+1}$, $\delta_n(x)$ is called the *parent* of x, and x is called a *child* of $\delta_n(x)$. Observe that trees may have infinite height, but if all T_n's are finite, the tree is *finitely branching*.

An *L-labelled tree* is a tree T together with a function $l : \bigsqcup_{n\in\mathbb{N}} T_n \longrightarrow L$. The definitions of *SLD*-tree and proof tree, Definitions 2.12 and 2.13 respectively, are of finitely branching labelled trees. Both satisfy a further property: for any node x, the children of x, i.e. the elements of $\delta^{-1}(x)$, have distinct labels. This reflects the definition of a logic program, following [35], as a set of clauses rather than as a list, and the distinctness of atoms in the body of a clause. We accordingly say an L-labelled tree is *locally injective* if for any node x, the children of x have distinct labels. Given a set L of labels, we denote the set of finitely branching locally injective L-labelled trees by $Tree_L$.

We briefly recall fundamental constructs of coalgebra, see also [22].

DEFINITION 3.1
For any endofunctor $H : C \longrightarrow C$, an *H-coalgebra* consists of an object X of C together with a map $x : X \longrightarrow HX$. A *map* of H-coalgebras from (X,x) to (Y,y) is a map $f : X \longrightarrow Y$ in C such that the diagram

$$\begin{array}{ccc} X & \longrightarrow & Y \\ \downarrow & & \downarrow \\ HX & \longrightarrow & HY \end{array}$$

commutes.

H-coalgebras and maps of H-coalgebras form a category *H-coalg*, with composition determined by that in C, together with a forgetful functor $U : H\text{-}coalg \longrightarrow C$, taking an H-coalgebra (X,x) to X.

EXAMPLE 3.2
Let P_f denote the endofunctor on *Set* that sends a set X to the set of its finite subsets, and sends a function $h : X \longrightarrow Y$ to the function $P_f(h) : P_f(X) \longrightarrow P_f(Y)$ sending a subset A of X to its image $f(A)$ in Y. A P_f-coalgebra (X,x) is a finitely branching transition system, one of the leading examples of coalgebra [22].

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

For any set L, the set $Tree_L$ of finitely branching locally injective L-labelled trees possesses a canonical P_f-coalgebra structure on it, sending (T,l) to the set of L-labelled trees determined by the children of the root of T. With mild overloading of notation, we denote this P_f-coalgebra by $Tree_L$.

THEOREM 3.3
The functor $U : P_f\text{-}coalg \longrightarrow Set$ has a right adjoint sending any set L to $Tree_L$.

PROOF. We have already seen that for any set L, the set $Tree_L$ possesses a canonical P_f-coalgebra structure given by sending an L-labelled tree (T,l) to the set of labelled trees determined by the children of the root of T.

For the universal property, suppose we are given a P_f-coalgebra (X,x) and a function $h : X \longrightarrow L$. Put $h_0 = h : X \longrightarrow L$. For any $a \in X$, $x(a)$ is a finite subset of X. So $P_f(h_0)(x(a))$ is a finite subset of L. Send a to the tree generated as follows: the root is labelled by $h_0(a)$; it has $P_f(h_0)(x(a))$ children, each labelled by the corresponding element of $P_f(h_0)(x(a))$; replace $h_0 : X \longrightarrow L$ by $h_1 = P_f(h_0)(x(-)) : X \longrightarrow P_f(L)$, and continue inductively.

The unicity of this as a map of coalgebras is determined by its construction together with the local injectivity condition; its well-definedness follows from the finiteness of any element of $P_f(X)$. ■

We adapt this analysis to give a semantic account of the way in which a logic program generates a tree of computations.

Given a set L of labels, an *L-labelled $\&\vee$-tree* is a finitely branching tree T together with a function $l : \bigsqcup_{n\in\mathbb{N}} T_{2n} \longrightarrow L$. In an L-labelled $\&\vee$-tree, the nodes of even height are called &-nodes, and the nodes of odd height are called $\vee$-nodes. So the &-nodes, such as the root, are labelled, while the $\vee$-nodes are not.

The and-or parallel derivation trees of Definition 2.16 are labelled $\&\vee$-trees satisfying an additional property that reflects logic programs consisting of sets rather than lists of clauses and the distinctness of atoms in the body of a clause. We express the condition semantically as follows: an L-labelled $\&\vee$-tree is *locally injective* if the children of any $\vee$-node have distinct labels, and if, for any two distinct children of an &-node, the sets of labels of their children are distinct (but may have non-trivial intersection), i.e. for any x, for any $y, z \in \delta^{-1}(x)$, one has $l(\delta^{-1}(y)) \neq l(\delta^{-1}(z))$. Given a set L of labels, we denote the set of locally injective L-labelled $\&\vee$-trees by $\&\vee\text{-}Tree_L$.

For any set L, the set $\&\vee\text{-}Tree_L$ has a canonical P_fP_f-coalgebra structure on it, sending (T,l) to the set of sets of labelled $\&\vee$-trees given by the set of sets of L-labelled $\&\vee$-trees determined by the children of each child of the root of T. Again, we overload notation, using $\&\vee\text{-}Tree_L$ to denote this coalgebra.

THEOREM 3.4
The functor $U : P_fP_f\text{-}coalg \longrightarrow Set$ has a right adjoint sending any set L to $\&\vee\text{-}Tree_L$.

PROOF. A proof is given by a routine adaption of the proof of Theorem 3.3. ■

There are assorted variants of Theorem 3.4. We shall need one for L-labelled $\&\vee_c$-trees, an L-labelled $\&\vee_c$-tree being the generalization of L-labelled $\&\vee$-tree given by allowing countable branching at even heights, i.e. allowing the root to have countably many children, but each child of the root to have only finitely many children, etc. Letting P_c denote the functor sending a set X to the set of its countable subsets, we have the following result.

THEOREM 3.5
The functor $U : P_cP_f\text{-}coalg \longrightarrow Set$ has a right adjoint sending any set L to $\&\vee_c\text{-}Tree_L$.

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

3.2 Coalgebraic semantics for ground programs

Using our coalgebraic calculus of trees, we now make precise, in the ground case, the relationship between logic programs and Gupta et al.'s and-or parallel derivation trees of Definition 2.16.

In general, if $U : H\text{-}coalg \longrightarrow C$ has a right adjoint G, the composite functor $UG : C \longrightarrow C$ possesses the canonical structure of a *comonad* $C(H)$, called the *cofree* comonad on H. A *coalgebra* for a comonad is subtly different to a coalgebra for an endofunctor as the former must satisfy two axioms, see also [4, 34]. We denote the category of $C(H)$-coalgebras by $C(H)$-$Coalg$.

THEOREM 3.6
[22] For any endofunctor $H : C \longrightarrow C$ for which the forgetful functor $U : H\text{-}coalg \longrightarrow C$ has a right adjoint, the category H-$coalg$ is canonically isomorphic to the category $C(H)$-$Coalg$. The isomorphism commutes with the forgetful functors to C.

Theorem 3.6 implies that for any H-coalgebra (X, x), there is a unique $C(H)$-coalgebra structure corresponding to it on the set X.

Recall from the Introduction that, in the ground case, a logic program can be identified with a coalgebra for the endofunctor $P_f P_f$ on Set. By Theorem 3.4, the forgetful functor $U : P_f P_f\text{-}coalg \longrightarrow Set$ has a right adjoint taking a set L to the coalgebra $\&\vee\text{-}Tree_L$. Thus the cofree comonad $C(P_f P_f)$ on $P_f P_f$ sends the set L to the set $\&\vee\text{-}Tree_L$.

So Theorem 3.6 tells us that every ground logic program P seen as a $P_f P_f$-coalgebra induces a canonical $C(P_f P_c)$-coalgebra structure on the set At of atoms underlying P, i.e. a function from At to $\&\vee\text{-}Tree_{At}$.

THEOREM 3.7
Given a $P_f P_f$-coalgebra $p : At \longrightarrow P_f P_f(At)$, the corresponding $C(P_f P_f)$-coalgebra has underlying set At and action $\bar{p} : At \longrightarrow \&\vee\text{-}Tree_{At}$ as follows:

For $A \in At$, the root of the tree $\bar{p}(A)$ is labelled by A. If $p(A) \in P_f P_f(At)$ consists of n subsets of $P_f(At)$, then the root of $\bar{p}(A)$ has n children. The number and labels of each child of each of those n children are determined by the number and choice of elements of At in the corresponding subset of $P_f(A)$. Continue inductively.

PROOF. In general, for any endofunctor H for which the forgetful functor $U : H\text{-}coalg \longrightarrow C$ has a right adjoint G, the $C(H)$-coalgebra induced by an H-coalgebra (X, x) is given as follows: $U(X, x) = X$, so the identity map $id : X \longrightarrow X$ can be written as $id : U(X, x) \longrightarrow X$. By the definition of adjoint, it corresponds to a map of the form $\epsilon_{(X,x)} : (X, x) \longrightarrow GX$. Applying U to $\epsilon_{(X,x)}$ gives the requisite coalgebra map $U\epsilon_{(X,x)} : X \longrightarrow C(H)X$.

Applying this to $H = P_f P_f$, this $C(P_f P_f)$-coalgebra structure is determined by the construction in the proof of Theorem 3.4, which is rewritten as the assertion of this theorem. ■

Comparing Theorem 3.7 with Definition 2.16, subject to minor reorganization, given a logic program P seen as a $P_f P_f$-coalgebra, the corresponding $C(P_f P_f)$-coalgebra structure on At sends an atom A to Gupta et al.'s and-or parallel derivation tree, characterizing their construction in the ground case.

EXAMPLE 3.8
Consider the ground logic program

```
q(b,a) ←
s(a,b) ←
```

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

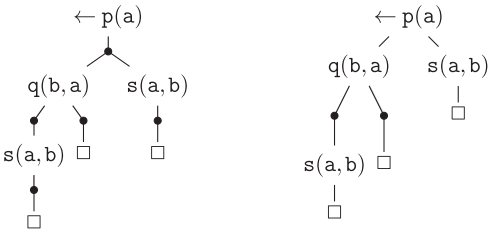

FIGURE 3. The action of $\overline{p}$:At $\longrightarrow C(P_fP_f)$(At) on `p(a)` and the corresponding and-or parallel derivation tree.

```
p(a) ← q(b,a), s(a,b)
q(b,a) ← s(a,b)
```

The program has three atoms, namely `q(b,a)`, `s(a,b)` and `p(a)`.
So $At = \{$`q(b,a)`, `s(a,b)`, `p(a)`$\}$. The program can be identified with the P_fP_f-coalgebra structure on At given by
$p($`q(b,a)`$) = \{\{\}, \{$`s(a,b)`$\}\}$, where $\{\}$ is the empty set.
$p($`s(a,b)`$) = \{\{\}\}$, i.e. the one element set consisting of the empty set.
$p($`p(a)`$) = \{\{$`q(b,a), s(a,b)`$\}\}$.

The corresponding $C(P_fP_f)$-coalgebra sends `p(a)` to the parallel refutation of `p(a)` depicted on the left side of Figure 3. Note that the nodes of the tree alternate between those labelled by atoms and those labelled by •. The set of children of each • represents a goal, made up of the conjunction of the atoms in the labels. An atom with multiple children is the head of multiple clauses in the program: its children represent these clauses. We use the traditional notation □ to denote $\{\}$.

Where an atom has a single •-child, we can elide that node without losing any information; the result of applying this transformation to our example is shown on the right side of Figure 3. The resulting tree is precisely the and-or parallel derivation tree for the atomic goal ←`p(a)`.

3.3 Coalgebraic semantics for arbitrary programs

Extending from ground logic programs to first-order programs is not routine. Following normal category theoretic practice, we model the first-order language underlying a logic program by a Lawvere theory [2, 6, 8].

DEFINITION 3.9
Given a signature Σ of function symbols, the *Lawvere theory* $\mathcal{L}_\Sigma$ generated by Σ is the following category: `ob`$(\mathcal{L}_\Sigma)$ is the set of natural numbers. For each natural number n, let $x_1, \ldots, x_n$ be a specified list of distinct variables. Define `ob`$(\mathcal{L}_\Sigma)(n,m)$ to be the set of m-tuples $(t_1, \ldots, t_m)$ of terms generated by the function symbols in Σ and variables $x_1, \ldots, x_n$. Define composition in $\mathcal{L}_\Sigma$ by substitution.

One can describe $\mathcal{L}_\Sigma$ without the need for a specified list of variables for each n: in a term t, a variable context is always implicit, i.e. $x_1, \ldots, x_m \vdash t$, and the variable context may be considered as a binder.

For each signature Σ, we extend the set At of atoms for a ground logic program to the functor $At : \mathcal{L}_\Sigma^{op} \to Set$ that sends a natural number n to the set of all atomic formulae generated

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

by Σ, variables among a fixed set $x_1,\ldots,x_n$, and the predicate symbols appearing in the logic program. A map $f:n\to m$ in $\mathcal{L}_\Sigma$ is sent to the function $At(f):At(m)\to At(n)$ that sends an atomic formula $A(x_1,\ldots,x_m)$ to $A(f_1(x_1,\ldots,x_n)/x_1,\ldots,f_m(x_1,\ldots,x_n)/x_m)$, i.e. $At(f)$ is defined by substitution.

Given a logic program P with function symbols in Σ, we would like to model P by the putative $[\mathcal{L}_\Sigma^{op},P_fP_f]$-coalgebra $p:At\longrightarrow P_fP_fAt$ on the category $[\mathcal{L}_\Sigma^{op},Set]$ whose n-component takes an atomic formula $A(x_1,\ldots,x_n)$ with at most n variables, considers all substitutions of clauses in P whose head agrees with $A(x_1,\ldots,x_n)$, and gives the set of sets of atomic formulae in antecedents. Unfortunately, it does not work.

Consider the logic program `ListNat` of Example 2.6. There is a map in $\mathcal{L}_\Sigma$ of the form $0\to 1$ that models the nullary function symbol 0. Naturality of the map $p:At\longrightarrow P_fP_fAt$ in $[\mathcal{L}_\Sigma^{op},Set]$ yields commutativity of the diagram

$$\begin{array}{ccc} At(1) & \longrightarrow & P_fP_fAt(1) \\ \downarrow & & \downarrow \\ At(0) & \longrightarrow & P_fP_fAt(0) \end{array}$$

There being no clause of the form `nat(x)` $\leftarrow$ in `ListNat`, commutativity implies that there cannot be a clause in `ListNat` of the form `nat(0)` $\leftarrow$ either, but in fact there is one.

We resolve this by relaxing the naturality condition on p to a subset condition, yielding lax naturality. To define it, we extend $At:\mathcal{L}_\Sigma^{op}\to Set$ to have codomain $Poset$, which we do by composing At with the inclusion of Set into $Poset$. Mildly overloading notation, we denote the composite by $At:\mathcal{L}_\Sigma^{op}\to Poset$.

$Poset$ canonically possesses the structure of a locally ordered category, i.e. there is a canonical partial order on each homset $Poset(P,Q)$ and it is respected by composition. It is given pointwise: $f\le g$ if and only if for all $x\in P$, one has $f(x)\le g(x)$ in Q. The category $\mathcal{L}_\Sigma$ also has a canonical locally ordered structure given by the discrete structure, i.e. $f\le g$ if and only if $f=g$. Any functor from $\mathcal{L}_\Sigma^{op}$ to $Poset$ is trivially locally ordered, i.e. preserves the partial orders.

DEFINITION 3.10
Given locally ordered functors $H,K:D\longrightarrow C$, a *lax natural transformation* from H to K is the assignment to each object d of D, of a map $\alpha_d:Hd\longrightarrow Kd$ such that for each map $f:d\longrightarrow d'$ in D, one has $(Kf)(\alpha_d)\le(\alpha_{d'})(Hf)$.

Locally ordered functors and lax natural transformations, with pointwise composition and pointwise ordering, form a locally ordered category we denote by $Lax(D,C)$.

A final problem arises in regard to the finiteness of the outer occurrence of P_f in P_fP_f. The problem is that substitution can generate infinitely many instances of clauses with the same head. For instance, if one extends `ListNat` with a clause of the form $A\leftarrow$ `nat(x)` with no occurrences of x in A, substitution yields the clause $A\leftarrow$ `nat(`$\mathtt{s}^n$`(0))` for every natural number n, giving rise to a countable set of clauses with head A. Graph connectivity, `GC`, gives another example, see Example 3.19.

We address this issue by replacing P_fP_f by P_cP_f, where P_c is the countable powerset functor, extending P_cP_f from Set to a locally ordered endofunctor on $Poset$, upon which composition yields the locally ordered endofunctor we seek on $Lax(\mathcal{L}_\Sigma^{op},Poset)$.

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

DEFINITION 3.11
Define $P_f : Poset \longrightarrow Poset$ by letting $P_f(P)$ be the partial order given by the set of finite subsets of P, with $A \leq B$ if for all $a \in A$, there exists $b \in B$ for which $a \leq b$ in P, with behaviour on maps given by image. Define P_c similarly but with countability replacing finiteness.

A cofree comonad $C(P_cP_f)$ exists on P_cP_f and, by Theorem 3.5, we can describe it: $C(P_cP_f)(P) = \&\vee_c\text{-}Tree_P$, with partial order structure generated by Definition 3.11. In order to extend the correspondence between P_cP_f-coalgebras $p : At \longrightarrow P_cP_fAt$ and $C(P_cP_f)$-coalgebras $\bar{p} : At \longrightarrow C(P_cP_f)At$ from *Poset* to $Lax(\mathcal{L}_\Sigma^{op}, Poset)$, we need to do some abstract category theory.

Let H be an arbitrary locally ordered endofunctor on an arbitrary locally ordered category C. Denote by $H\text{-}coalg_{oplax}$ the locally ordered category whose objects are H-coalgebras and whose maps are oplax maps of H-coalgebras, meaning that, in the square

$$\begin{array}{ccc} X & \longrightarrow & Y \\ \downarrow & \leq & \downarrow \\ HX & \longrightarrow & HY \end{array}$$

the composite via HX is less than or equal to the composite via Y. Since C and H are arbitrary, one can replace C by $Lax(D, C)$, for any category D; and replace H by $Lax(D, H)$, yielding the locally ordered category $Lax(D, H)\text{-}coalg_{oplax}$.

PROPOSITION 3.12
The locally ordered category $Lax(D, H)\text{-}coalg_{oplax}$ is canonically isomorphic to $Lax(D, H\text{-}coalg_{oplax})$.

PROOF. Unwinding the definitions, to give a functor $J : D \longrightarrow H\text{-}coalg_{oplax}$ is, by definition, to give, for each object d of D, a map in C of the form $Jd : J_0d \longrightarrow HJ_0d$, and, for each map $f : d \longrightarrow d'$ in D, a map in C of the form $J_0f : J_0d \longrightarrow J_0d'$, such that

$$\begin{array}{ccc} J_0d & \xrightarrow{J_0f} & J_0d' \\ {\scriptstyle Jd}\downarrow & \leq & \downarrow{\scriptstyle Jd} \\ HJ_0d & \xrightarrow[HJ_0f]{} & HJ_0d' \end{array}$$

subject to locally ordered functoriality equations.

These data and axioms can be re-expressed as a locally ordered functor $J_0 : D \longrightarrow C$ together with a lax natural transformation $j : J_0 \longrightarrow HJ_0$, the condition for lax naturality of j in regard to the map f in D being identical to the condition that J_0f be an oplax map of coalgebras from Jd to Jd'.

This yields a canonical bijection between the sets of objects of $Lax(D, H\text{-}coalg_{oplax})$ and $Lax(D, H)\text{-}coalg_{oplax}$, that bijection canonically extending to a canonical isomorphism of locally ordered categories. ■

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

PROPOSITION 3.13
Given a locally ordered comonad G on a locally ordered category C, the data given by $Lax(D,G)$: $Lax(D,C) \to Lax(D,C)$ and pointwise liftings of the structural natural transformations of G yield a locally ordered comonad we also denote by $Lax(D,G)$ on $Lax(D,C)$.

PROOF. This holds by tedious but routine checking of all the axioms in the definition of locally ordered comonad. ■

Given a locally ordered comonad G, denote by $G\text{-}Coalg_{oplax}$ the locally ordered category whose objects are G-coalgebras and whose maps are oplax maps of G-coalgebras.

PROPOSITION 3.14
Given a locally ordered comonad G, $Lax(D,G)\text{-}Coalg_{oplax}$ is canonically isomorphic to $Lax(D, G\text{-}Coalg_{oplax})$.

PROOF. A proof is given by routine extension of the proof of Proposition 3.12. ■

THEOREM 3.15
[25] Given a locally ordered endofunctor H on a locally ordered category with finite colimits C, if $C(H)$ is the cofree comonad on H, then $H\text{-}coalg_{oplax}$ is canonically isomorphic to $C(H)\text{-}Coalg_{oplax}$.

Combining Proposition 3.12, Proposition 3.14 and Theorem 3.15, we can conclude the following:

THEOREM 3.16
Given a locally ordered endofunctor H on a locally ordered category with finite colimits C, if $C(H)$ is the cofree comonad on H, then there is a canonical isomorphism

$$Lax(D,H)\text{-}coalg_{oplax} \simeq Lax(D,C(H))\text{-}Coalg_{oplax}$$

COROLLARY 3.17
For any locally ordered endofunctor H on *Poset*, if $C(H)$ is the cofree comonad on H, then there is a canonical isomorphism

$$Lax(\mathcal{L}_\Sigma^{op},H)\text{-}coalg_{oplax} \simeq Lax(\mathcal{L}_\Sigma^{op},C(H))\text{-}Coalg_{oplax}$$

Putting $H = P_c P_f$, Corollary 3.17 gives us the abstract result we need. The lax natural transformation $p : At \longrightarrow P_c P_f At$ generated by a logic program P, evaluated at a natural number n, sends an atomic formula $A(x_1,\ldots,x_n)$ to the set of sets of antecedents in substitution instances of clauses in P for which the head of the substituted instance agrees with $A(x_1,\ldots,x_n)$. That in turn yields a lax natural transformation $\bar{p} : At \longrightarrow C(P_c P_f)At$, which, evaluated at n, is the function from the set $At(n)$ to the set $\&\vee_c\text{-}Tree_{At(n)}$ determined by the construction of Theorem 3.7 if one treats the variables $x_1,\ldots,x_n$ as constants. See also [7] for a Laxness-free semantics for CoALP.

EXAMPLE 3.18
Consider `ListNat` as in Example 2.6. Suppose we start with $A(x,y) \in At(2)$ given by the atomic formula `list(cons(x,cons(y,x)))`. Then $\bar{p}(A(x,y))$ is the element of $C(P_c P_f)At(2) = \&\vee_c\text{-}Tree_{At(2)}$ expressible by the tree on the left-hand side of Figure 4. This tree agrees with the start of the and-or parallel derivation tree for `list(cons(x,cons(y,x)))`. It has leaves `nat(x)`, `nat(y)` and `list(x)`, whereas the and-or parallel derivation tree follows those nodes, using substitutions determined by mgu's that might not be consistent with each other, e.g. there is no consistent substitution for `x`.

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

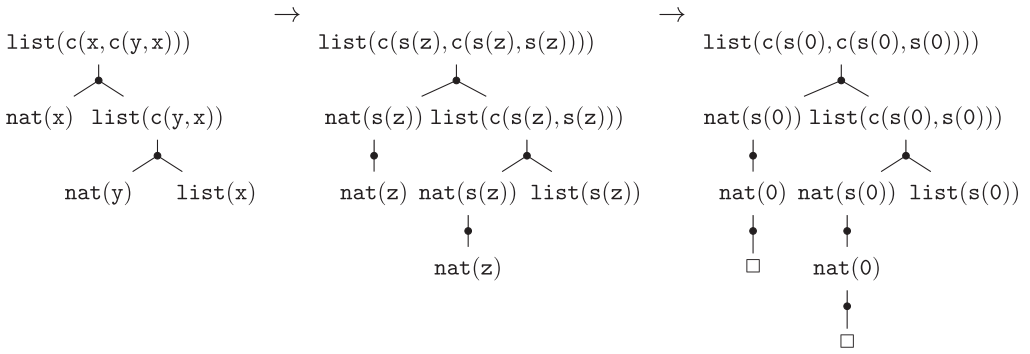

FIGURE 4. The left-hand tree depicts $\bar{p}$(list(cons(x,cons(y,x)))), the middle tree depicts $\bar{p}At(s,s)$(list(cons(x,cons(y,x)))), i.e. $\bar{p}$(list(cons(s(z),cons(s(z),s(z))))), and the right tree depicts $\bar{p}At(0)At(s,s)$(list(cons(x,cons(y,x)))); cons is abbreviated by c.

Lax naturality means a substitution potentially yields two different trees: one given by substitution into the tree, then pruning to remove redundant branches, the other given by substitution into the root, then applying $\bar{p}$.

For example, we can substitute $s(z)$ for both x and y in list(cons(x,cons(y,x))). This substitution is given by applying At to the map $(s,s):1 \longrightarrow 2$ in $\mathcal{L}_\Sigma$. So $At((s,s))(A(x,y))$ is an element of $At(1)$. Its image under $\bar{p}(1):At(1) \longrightarrow C(P_cP_f)At(1)$ is the element of $C(P_cP_f)At(1) = \&\vee_c\text{-}Tree_{At(1)}$ given by the tree in the middle of Figure 4.

The laxness of the naturality of $\bar{p}$ is indicated by the increased length, in two places, of this tree. Before those two places, the two trees have the same structure.

Now suppose we make the further substitution of 0 for z. This substitution is given by applying At to the map $0:0 \to 1$ in $\mathcal{L}_\Sigma$. In Figure 4, we depict $\bar{p}(0)At(0)At((s,s))(A(x,y)) \in \&\vee_c\text{-}Tree_{At(1)}$ on the right. Two of the leaves of the latter tree are labelled by □, but one leaf, namely list(s(0)) is not, so the tree does not yield a proof. Again, observe the laxness.

This requires care. Consider the following example, studied in [46].

EXAMPLE 3.19 (GC)
Let GC (for graph connectivity) denote the logic program

```
connected(x,x) ←
connected(x,y) ← edge(x,z),connected(z,y)
```

There may be additional function symbols, such as a unary s, and additional clauses to give a database, such as edge(0,s(0)) ← and edge(s(0),s(s(0))) ← . Note the presence of a variable z in the body but not the head of the clause

```
connected(x,y) ← edge(x,z),connected(z,y)
```

That allows derivations involving infinitely many variables, thus not directly yielding a subtree of $\bar{p}$(connected(x,y)) $\in \&\vee_c\text{-}Tree_{At(n)}$ for any n.

The subtle relationship between the finite and the infinite illustrated by Example 3.19 is fundamental to the idea of coalgebraic logic programming, which we develop in the latter sections of the article. See also Figure 6.

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

DEFINITION 3.20
Let P be a logic program, G be an atomic goal, and T be the $\&\vee_c$-tree determined by P and $G \in At(n)$. A subtree T' of T is called a *derivation subtree* of T if it satisfies the following conditions:

- the root of T' is the root of T (up to variable renaming);
- if an and-node belongs to T', then at most one of its children belongs to T'.
- if an or-node belongs to T', then all its children belong to T'.

A finite derivation tree is *successful* if its leaves are all or-nodes (equivalently, they are followed only by $\Box$ in the usual pictures).

By Example 3.19, derivations need not directly yield derivation subtrees. Nevertheless, all subderivations of finite length of a derivation do form derivation subtrees.

THEOREM 3.21 (Soundness and Completeness of *SLD*-refutations)
Let P be a logic program, and G be an atomic goal.

(1) Soundness. If there is an *SLD*-refutation for G in P with computed answer θ, then for some n with $G\theta \in At(n)$, the $\&\vee_c$-tree for $G\theta$ contains a successful derivation subtree.
(2) Completeness. If the $\&\vee_c$-tree for $G\theta \in At(n)$ contains a successful derivation subtree, then there exists an *SLD*-refutation for G in P, with computed answer λ for which $\lambda\sigma = \theta$ for some σ.

PROOF. The finiteness of refutations makes this a routine adaptation of the soundness and completeness of the collectivity of *SLD*-trees for *SLD*-refutation. ■

3.4 *Coalgebraic semantics and the theory of observables*

Our coalgebraic analysis relates closely to the *Theory of Observables* for logic programming developed in [9]. In that theory, the traditional characterization of logic programs in terms of input/output behaviour and successful derivations is not sufficient for the purposes of program analysis and optimization. One requires more complete information about *SLD*-derivations, specifically the sequences of goals and most general unifiers used. Infinite derivations can be meaningful. The following observables are critical to the theory [9, 14].

DEFINITION 3.22
(1) A *call pattern* is a sequence of atoms selected in an *SLD*-derivation; a *correct call pattern* is a sequence of atoms selected in an *SLD*-refutation.
(2) A *partial answer* is a substitution associated with a resolvent in an *SLD*-derivation; a *correct partial answer* is a substitution associated with a resolvent in an *SLD*-refutation.

As explained in [9, 14], semantics of logic programs aims to identify observationally equivalent logic programs and to distinguish logic programs that are not observationally equivalent. So the definitions of observation and semantics are interdependent. Observational equivalence was defined in [14] as follows.

DEFINITION 3.23
Let P_1 and P_2 be logic programs with the same alphabet. Then P_1 is *observationally equivalent* to P_2, written $P_1 \approx P_2$, if for any goal G, the following conditions hold:

(1) G has an *SLD*-refutation in P_1 if and only if G has an *SLD*-refutation in P_2.
(2) G has the same set of computed answers in P_1 as in P_2.

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

(3) G has the same set of (correct) call patterns in P_1 as in P_2.
(4) G has the same set of (correct) partial answers in P_1 as in P_2.

THEOREM 3.24 (Correctness)
For logic programs P_1 and P_2, if the $Lax(\mathcal{L}_\Sigma^{op}, C(P_c P_f))$-coalgebra structure $\bar{p}_1$ generated by P_1 is isomorphic to the $Lax(\mathcal{L}_\Sigma^{op}, C(P_c P_f))$-coalgebra structure $\bar{p}_2$ generated by P_2 (denoted $\bar{p_1} \cong \bar{p_2}$), then $P_1 \approx P_2$.

The converse of Theorem 3.24, *full abstraction*, does not hold, i.e. with the above definition of observational equivalence, there are observationally equivalent programs that have different $\&\vee_c$-*Tree*s.

EXAMPLE 3.25
Consider logic programs P_1 and P_2 with the same clauses except for one: P_1 contains $A \leftarrow B_1, \texttt{false}, B_2$; and P_2 contains the clause $A \leftarrow B_1, \texttt{false}$ instead. The atoms in the clauses are such that B_1 has a refutation in P_1 and P_2, and `false` is an atom that has no refutation in the programs. In this case, assuming a left-to-right sequential evaluation strategy, all derivations that involve the two clauses in P_1 and P_2 will always fail on `false`, and P_1 will be observationally equivalent to P_2, but they generate different trees because of B_2.

We can recover full abstraction if we adapt Definitions 3.22 and 3.23 so that they do not rely upon an algorithm to choose a selected atom but rather allow arbitrary choices. This is typical of coalgebra, yielding essentially an instance of bisimulation [22]. In order to do that, we need to modify Definitions 2.8 and 2.9 to eliminate the algorithm used in the definitions leading to *SLD*-derivations.

DEFINITION 3.26
Let a goal G be $\leftarrow A_1, \ldots, A_k$ and a clause C be $A \leftarrow B_1, \ldots, B_q$. Then G' is *non-deterministically derived* from G and C using mgu θ if the following conditions hold:

- θ is an mgu of some atom A_m in the body of G and A;
- G' is the goal $\leftarrow (A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta$.

Definition 3.26 differs from Definition 2.8 in precisely one point: the former refers to 'some atom' where the latter refers to 'the selected atom', with the selection being determined by an algorithm. The distinction means that Definition 3.26 has nondeterminism built into the choice of atom, which in turn implies the possibility of parallelism in implementation. We will exploit that later. It further implies that a verbatim restatement of Definition 3.23 but with '*SLD*-derivation' replaced by 'coinductive derivation' also implies the possibility of implementation based on parallelism.

DEFINITION 3.27
A *non-deterministic derivation* of $P \cup \{G\}$ consists of a sequence of goals $G = G_0, G_1, \ldots$ called *non-deterministic resolvents*, a sequence $C_1, C_2, \ldots$ of variants of program clauses of P, and a sequence $\theta_1, \theta_2, \ldots$ of mgu's such that each G_{i+1} is derived from G_i using θ_{i+1}. A *non-deterministic refutation* of $P \cup \{G\}$ is a finite non-deterministic derivation of $P \cup \{G\}$ such that its last goal is empty, denoted by $\square$. If $G_n = \square$, we say that the refutation has length n. The composite $\theta_1 \theta_2 \ldots$ is called a *computed answer*.

Figure 5 exhibits a non-deterministic derivation for the goal $G = \texttt{stream(x)}$ and the program `Stream` from Example 2.5.

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

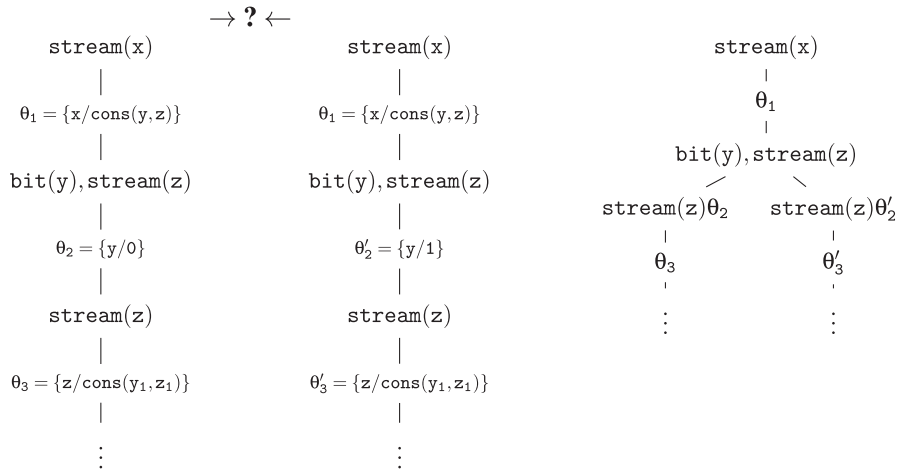

FIGURE 5. Left: Two possible choices for non-deterministic derivation for the goal $G=$ `stream(x)` and the program `Stream`, with $\theta_1=\{x/scons(y,z)\}$, $\theta_2=\{y/0\}$, $\theta_2'=\{y/1\}$ and $\theta_3=\theta_3'=\{y/scons(y_1,z_1)\}$. Right: the two non-deterministic derivations shown in the form of an SLD-tree.

Given logic programs P_1 and P_2 over the same alphabet, we write $P_1 \approx_n P_2$ if, consistently replacing *SLD*-derivation and *SLD*-refutation by non-deterministic derivation and non-deterministic refutation in Definitions 3.22 and 3.23, P_1 and P_2 are observationally equivalent.

THEOREM 3.28 (Full abstraction)
For any logic programs P_1 and P_2 with the same alphabet, $P_1 \approx_n P_2$ if and only if $\bar{p}_1 \cong \bar{p}_2$.

PROOF. This is routine: as we have allowed any choice of atom rather than depending upon an algorithm to choose a selected atom, observational equivalence accounts for all branches. ■

The way in which coalgebra models non-deterministic derivations is necessarily complex for a few reasons:

(1) A non-deterministic derivation might involve infinitely many variables, but each $At(n)$ only allows for a finite number of variables.
(2) A non-deterministic derivation could involve an infinite chain of substitutions, but an element of $At(n)$ does not allow for that. Consider e.g. Example 2.11.

So, within coalgebra, one can only give a chain of finite approximants to a non-deterministic derivation. Theorem 3.21 extends routinely from *SLD*-refutations to non-deterministic refutations. We can further extend it to non-deterministic derivations too, with due care for the possibility of derivations involving infinitely many variables as induced by Example 3.19.

THEOREM 3.29 (Soundness and Completeness of non-deterministic derivations)
Let P be a logic program, with p its induced $Lax(\mathcal{L}_\Sigma^{op}, P_c P_f)$-coalgebra, and let G be an atomic goal.

(1) Soundness. Given any finite subderivation of a non-deterministic derivation of $P \cup \{G\}$ with partial answer θ, the subderivation generates a derivation subtree of $\bar{p}(G\theta)$ for some n with $G\theta \in At(n)$.

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

(2) Completeness. Given a list $\theta_0, \theta_1, \ldots$ of substitutions, and a list $T_0, T_1, \ldots$ of finite derivation subtrees of $\bar{p}(G\theta_0)$, $\bar{p}(G\theta_0\theta_1)$, etcetera, with $T_n\theta_n$ a subtree of T_{n+1} for each n, there is a non-deterministic derivation of $P \cup \{G\}$ that generates the T_n's.

PROOF. The soundness claim follows from induction on the length of a finite subderivation. For length 0, the statement is trivial. Assume it is true for length n, with derivation subtree T_n of $\bar{p}(G\theta)$. Suppose G_{n+1} is derived from G_n using θ_{n+1} and clause C_{n+1}, with respect to the atom A_m in G_n. Apply θ_{n+1} to the whole of T_n, yielding a derivation subtree of $\bar{p}(G\theta\theta_{n+1})$, and extend the tree at the leaf $A_m\theta_{n+1}$ by applying θ_{n+1} to each atom in the body of the C_{n+1} to provide the requisite *and*-nodes.

Completeness also holds by induction. For $n=0$, given a finite derivation subtree T_0 of $\bar{p}(G\theta_0)$, if follows from the finiteness of T_0 and the fact that it is a subtree of $\bar{p}(G\theta_0)$ that it can be built from a finite sequence of derivation steps starting from G, followed by a substitution.

Now assume that is the case for T_n, and we are given T_{n+1} subject to the conditions stated in the theorem. By our inductive hypothesis, we have a finite derivation from G, followed by a substitution, that yields the tree T_n. That is therefore also true for $T_n\theta_{n+1}$. As T_{n+1} is a finite extension of T_n and is a subtree of $\bar{p}(G\theta_0 \ldots \theta_{n+1})$, one can make a finite extension of the finite derivation from G that, followed by a substitution, yields T_{n+1}. ■

4 Corecursion in logic programming

We now move from abstract theory towards the development of coalgebraic logic programming. Central to this is the relationship between the finite and the infinite. We introduce a new kind of tree in order to make the subtle relationship precise and underpin our formulation of CoALP, a variant of logic programming based on our coalgebraic semantics.

4.1 Coinductive derivations

We first return to our running example of program `Stream`. In Section 2 and Figure 5, we have seen that this program gives rise to non-terminating SLD-derivations and infinite SLD-trees; moreover, the conventional greatest fixed point semantics is unsound for such cases. Coalgebraic semantics of Section 3 suggests the following tree-based semantics of derivations in `Stream`, see Figure 6. Comparing Figure 5 and Figure 6, we see that computations described by $\&\vee_c$-*Tree*s suggest parallel branching, much like and-or parallel trees [19], and also—finite height trees in the case of `Stream`. These two features will guide us in this Section, when we develop the computational algorithms for CoALP, and then follow them with implementation in Section 6.

We suggest the following definition of *coinductive tree* as a close computational counterpart of the $\&\vee_c$-*Tree*s of the previous section.

DEFINITION 4.1
Let P be a logic program and $G=A$ be an atomic goal. The *coinductive tree* for A is a possibly infinite tree T satisfying the following properties.

- A is the root of T.
- Each node in T is either an and-node or an or-node.
- Each or-node is given by $\bullet$.
- Each and-node is an atom.
- For every and-node A' occurring in T, if there exist exactly $m>0$ distinct clauses $C_1, \ldots, C_m$ in P (a clause C_i has the form $B_i \leftarrow B_1^i, \ldots, B_{n_i}^i$, for some n_i), such that $A' = B_1\theta_1 = \ldots = B_m\theta_m$, for

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

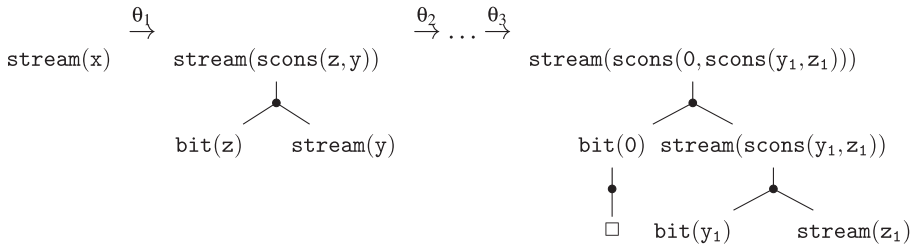

FIGURE 6. According to the coalgebraic semantics of the previous section, the left-hand tree depicts $\bar{p}(\texttt{stream(x)})$, the middle tree depicts $\bar{p}\ At(scons)(\texttt{stream(x)})$, and the right tree depicts $\bar{p}\ At(scons)At(0)At(scons)(\texttt{stream(x)})$. The same three trees represent a coinductive derivation for the goal $G = \texttt{stream(x)}$ and the program `Stream`, with $\theta_1 = x/scons(z,y)$, $\theta_2 = z/0$ and $\theta_3 = y/scons(y_1,z_1)$.

mgu's $\theta_1,\ldots,\theta_m$, then A' has exactly m children given by or-nodes, such that, for every $i \in m$, the ith or-node has n_i children given by and-nodes $B^i_1\theta_i,\ldots,B^i_{n_i}\theta_i$.

Three coinductive trees for program `Stream` are shown in Figure 6. In contrast to *SLD*-trees, coinductive trees restrict unification to term matching, i.e. we have $A' = B\theta$, rather than $A'\theta = B\theta$. Unification in general is inherently sequential [11], but term matching is parallelizable. At the same time, this restriction provides a powerful tool for implicit resource control as it allows one to unfold coinductive trees lazily, keeping each individual tree at a finite size, provided the program is well founded; as we discuss in detail in Section 4.2. In our implementation, we assume that every branch of the coinductive tree can by constructed in parallel to other branches, that is, no extra algorithm coordinating the variable substitutions is needed. See also Sections 5 and 6.

As can be seen from Figures 4 and 6, one coinductive tree T may not produce the answer corresponding to a refutation by the SLD-resolution. Instead, a sequence of coinductive trees may be needed to advance the derivation. We introduce a new derivation algorithm that allows proof search using coinductive trees. We modify the definition of a goal by taking it to be a pair $<A,T>$, where A is an atom, and T is the coinductive tree determined by A.

DEFINITION 4.2
Let G be a goal given by an atom $\leftarrow A$ and the coinductive tree T induced by A, and let C be a clause $H \leftarrow B_1,\ldots,B_n$. Then goal G' is *coinductively derived* from G and C using mgu θ if the following conditions hold:

- A' is an atom in T.
- θ is an *mgu* of A' and H.
- G' is given by the atom $\leftarrow A\theta$ and the coinductive tree $T\theta$.

Coinductive derivations resemble *tree rewriting*. They produce the 'lazy' corecursive effect: derivations are given by potentially infinite number of steps, where each individual step is executed in finite time.

EXAMPLE 4.3
Figure 5 shows how `Stream` gives rise to infinite *SLD*-trees. But it only gives rise to finite coinductive trees because of the term matching condition in the definition of coinductive tree. Moreover, there

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

is only one coinductive tree for any goal. An infinite derivation can be modelled by an infinite coinductive derivation, as illustrated in Figure 6.

EXAMPLE 4.4
`ListNat`, i.e. Example 2.6, also gives rise to infinite *SLD*-trees, see Figure 2, but it also only gives rise to finite coinductive trees as, again, all clauses in `ListNat` are guarded by constructors `0, s, nil, cons`. A coinductive derivation for `ListNat` and the goal `list(cons(x,cons(y,x)))` is illustrated in Figure 4. Again, an infinite derivation can be modelled by an infinite chain of finite coinductive trees.

Note that the definition of coinductive derivation allows for non-deterministic choice of the leaf atoms; compare e.g. with previously seen non-deterministic derivations from Definition 3.26. Transitions between coinductive trees can be done in a sequential or parallel manner. That is, if there are several non-empty leaves in a tree, any such leaf can be unified with some clause in P. Such leaves can provide substitutions for sequential or parallel tree transitions. In Figure 6, the substitution $\theta' = \theta_2\theta_3$ is derived by considering mgu's for two leaves in $G_1 = <\texttt{stream(scons(z,y))}, T_1>$; but, although two separate and-leaves were used to compute θ', θ' was computed by composing the two substitutions sequentially, and only one tree, T_3, was produced. However, we could concurrently derive two trees from T_2 instead, $G_2' = <\texttt{stream(scons(0,y))}, T_2>$ and $G_2'' = <\texttt{stream(scons(z,scons}(y_1,z_1)))\texttt{)}, T_2'>$. We exploit parallelism of such transitions in Sections 5 and 6.

DEFINITION 4.5
Let P be a logic program, G be an atomic goal, and T be a coinductive tree determined by P and G. A subtree T' of T is called a *coinductive subtree* of T if it satisfies the following conditions:

- the root of T' is the root of T (up to variable renaming);
- if an and-node belongs to T', then one of its children belongs to T'.
- if an or-node belongs to T', then all its children belong to T'.

A finite coinductive (sub)tree is called a *success (sub)tree* if its leaves are empty goals (equivalently, they are followed only by $\Box$ in the usual pictures).

Note that coinductive subtrees are not themselves coinductive trees: coinductive trees give account to all possible and-or-parallel proof choices given the terms determined by the goal, whereas a coinductive subtree corresponds to one possible sequential SLD-derivation for the given goal, where unification in the SLD-derivation is restricted to term-matching, cf. Definition 4.1.

In what follows, we will assume that the goal in Definition 4.2 is given by an atom $\leftarrow A$, and T is implicitly assumed. This convention agrees with the standard logic programming practice, where goals are given by first-order atoms. For example, we say that the goal `stream(x)` generates the coinductive derivation of Figure 6. The next definition formalizes this convention.

DEFINITION 4.6
A *coinductive derivation* of $P \cup \{G\}$ consists of a sequence of goals $G = G_0, G_1, \ldots$ called *coinductive resolvents* and a sequence $\theta_1, \theta_2, \ldots$ of mgu's such that each G_{i+1} is derived from G_i using θ_{i+1}. A *coinductive refutation* of $P \cup \{G\}$ is a finite coinductive derivation of $P \cup \{G\}$ such that its last tree contains a success subtree. If G_n contains a success subtree, we say that the refutation has length n.

We now modify Definitions 3.22 and 3.23 of observational equivalence. Suppose the definitions of a (correct) call pattern and a (correct) partial answer from Definition 3.22 are re-formulated with

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

respect to coinductive derivations, rather than SLD-derivations. Unlike SLD-derivations, coinductive derivations perform computations in 'two dimensions'—at the level of coinductive trees and at the level of transitions between coinductive trees. Both dimensions of computations can be observed. The next definition formalizes this.

DEFINITION 4.7
Let P_1 and P_2 be logic programs with the same alphabet. Then P_1 is *coinductively observationally equivalent* to P_2, written $P_1 \approx_c P_2$, if for any goal G, the following conditions hold:

1.-4. Conditions of Definition 3.23, but with coinductive derivations replacing SLD-derivations.
5. The coinductive tree T_1 for G and P_1 contains a coinductive subtree C iff the coinductive tree T_2 for G and P_2 contains C, modulo variable renaming.

For ground programs, all coinductive derivations will have length 0, and the coinductive tree generated for a given goal will account for all alternative derivations by SLD-resolution. Therefore, conditions [1.-4.] of coinductive observational equivalence will be trivially satisfied for all ground logic programs. However, condition [5.] will be able to distinguish different logic programs in such cases.

THEOREM 4.8 (Full abstraction)
For any logic programs P_1 and P_2 with the same alphabet, $P_1 \approx_c P_2$ if and only if $\bar{p}_1 \cong \bar{p}_2$.

PROOF. Similarly to Theorem 3.28, we allowed any choice of resolvents, and observational equivalence accounts for all branches. This accounts for conditions [1.-4.] in Definition 4.7.

For condition [5.] of coinductive observational equivalence, consider coinductive trees: their structure and labels account for all possible clauses that can be *matched* with the current goal and subgoals via mgu's. If, for any goal G with n distinct variables, P_1 and P_2 produce equivalent coinductive trees, then the image of G under $\bar{p}_1$ will be isomorphic to the image of G under $\bar{p}_2$.

The other direction is straightforward. ■

In general, the definition of the coinductive tree permits generation of coinductive trees containing infinitely many variables. So a coinductive tree for a goal A need not be a subtree of $\bar{p}(A) \in \&\vee_c\text{-}Tree_{At(n)}$ for any n. But every finite one must be, and establishment or otherwise of finiteness of coinductive trees is critical for us.

EXAMPLE 4.9
GC, i.e. Example 3.19, has a clause

```
connected(x,y) ← edge(x,z),connected(z,y)
```

in which there is a variable in the body but not the head. If one includes a unary function symbol s in GC, the clause induces infinite coinductive trees, all subtrees of $\bar{p}$(`connected(x,y)`)$\in \&\vee_c\text{-}Tree_{At(2)}$, as there are infinitely many possible substitutions for z. The clause also induces infinitely many coinductive trees that do not lie in $\bar{p}$(`connected(x,y)`)$\in \&\vee_c\text{-}Tree_{At(n)}$ for any n.

Note that, in Section 3, we established two different kinds of soundness and completeness results: one related the coalgebraic semantics to finite SLD-refutations (cf. Theorem 3.21), another—to potentially infinite non-deterministic derivations (cf. Theorem 3.29). The second theorem generalized the first. As we explain in the next section, one of the main advantages of CoALP is graceful handling

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

of corecursive programs and coinductive definitions. This is why we consider derivations of arbitrary size in our next statement of soundness and completeness for CoALP, as follows.

THEOREM 4.10 (Soundness and Completeness of coinductive derivations)
Let P be a logic program, with $\bar{p}$ its induced $Lax(\mathcal{L}_{\Sigma}^{op}, P_c P_f)$-coalgebra, and let G be an atomic goal.

(1) Soundness. Given a coinductive tree $\mathcal{T}$ resulting from a coinductive derivation of $P \cup \{G\}$ with partial answer θ, there is a coinductive subtree C of $\mathcal{T}$, such that C is a derivation subtree of $\bar{p}(G\theta)$ for some n for which $G\theta \in At(n)$.
(2) Completeness. Given a list $\theta_0, \theta_1, \ldots$ of substitutions, and a list $T_0, T_1, \ldots$ of finite derivation subtrees of $\bar{p}(G\theta_0)$, $\bar{p}(G\theta_0\theta_1)$, etcetera, with $T_n\theta_n$ a subtree of T_{n+1} for each n, there is a coinductive derivation of $P \cup \{G\}$, involving computed substitutions $\theta_0, \theta_1, \ldots$ and coinductive trees $\mathcal{T}_0, \mathcal{T}_1, \ldots$ such that, for each n, $\mathcal{T}_n$ contains a coinductive subtree C_n, such that C_n contains T_n, modulo variable renaming.

PROOF. Soundness. Consider a coinductive derivation of $P \cup \{G\}$ with partial answer $\theta = \theta_0, \theta_1, \ldots, \theta_k$: it contains a sequence of coinductive trees $\mathcal{T}_0, \mathcal{T}_1, \ldots, (\mathcal{T}_k = \mathcal{T})$ for $G\theta_0, G\theta_1, \ldots, G\theta_k$. Each $\mathcal{T}_i$ is uniquely determined by $G\theta_i$, although $\mathcal{T}_i$ may have infinite branches (cf. Example 2.11). In general case, $\mathcal{T}$ may contain several coinductive subtrees, each giving an account to one possible combination of clauses determining or-nodes. Consider one such coinductive subtree C, and suppose it contains n distinct variables. Then, by construction of $\&\vee_c$-$Tree_{At(n)}$ and Definition 3.20, there will be a derivation subtree in $\bar{p}(G\theta)$ corresponding to C.

Completeness. The proof is similar to the proof of Theorem 3.29, but here, we also note that each step in a coinductive derivation is given by a coinductive tree, rather than by a resolvent. The role of a non-deterministic SLD-derivation is now delegated to a coinductive subtree C_n of the coinductive tree $\mathcal{T}_n$. Note that coinductive trees may be finite for guarded clauses like `Stream` (cf. Example 4.3), and hence a sequence of coinductive trees $\mathcal{T}_1, \mathcal{T}_2, \ldots$ will yield all T_n's (cf. Theorem 3.29). However, non-guarded clauses (cf. Example 2.11) give rise to infinite coinductive trees, in which case T_n will be only a fragment of a coinductive subtree C_n of the coinductive tree $\mathcal{T}_n$. In that case, an infinite sequence of T_ns would approximate one C_n, similarly to Theorem 3.29. ■

Discussion of the constructive component of the completeness results for CoALP and the constructive reformulation of the above completeness theorem can be found in [32]. The problem of distinguishing cases with finite and infinite coinductive trees will be the main topic of the next section.

4.2 *Guarding corecursion*

In this section, we consider various methods used in logic programming to *guard* (co-) recursion, and introduce our own method for guarding corecursion in CoALP.

As Example 2.15 illustrates, *SLD*-derivations may yield looping infinite derivation chains for programs like stream. In *Coinductive Logic Programming* (Co-LP) [18, 45], such were addressed by introducing a procedure allowing one to terminate derivations with the flag '`stream(x)` is proven', whenever such a loop was detected. Extending this 'rule' to inductive computations leads to unsound results: in the inductive case, infinite loops normally indicate lack of progress in a derivation rather than 'success'. Thus, explicit annotation of predicates was required. Consider the following example.

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

EXAMPLE 4.11
The annotated logic program below comprises both inductive and coinductive clauses.

$$\begin{aligned}
\texttt{bit}^i(0) &\leftarrow \\
\texttt{bit}^i(1) &\leftarrow \\
\texttt{stream}^c(\texttt{scons}(x,y)) &\leftarrow \texttt{bit}^i(x), \texttt{stream}^c(y) \\
\texttt{list}^i(\texttt{nil}) &\leftarrow \\
\texttt{list}^i(\texttt{cons}(x,y)) &\leftarrow \texttt{bit}^i(x), \texttt{list}^i(y)
\end{aligned}$$

Only infinite loops produced for corecursive goals (marked by c) are gracefully terminated; others are treated as 'undecided' proof branches.

In practice, these annotations act as locks and keys in resource logics, allowing or disallowing infinite data structures. There are drawbacks:

★ some predicates may behave inductively or coinductively depending on the arguments provided, and such cases need to be resolved dynamically, not statically, in which case predicate annotation fails.

★★ the coinductive algorithm [18, 45] is not in essence a lazy infinite (corecursive) computation. Instead, it substitutes an infinite proof by a finite derivation, on the basis of guarantees of the data regularity in the corecursive loops. But such guarantees cannot always be given: consider computing the number π.

The coinductive derivations we introduced in the previous section give an alternative solution to the problem of guarding corecursion. We have already seen that Definition 4.1 determined finite coinductive trees both for the coinductive program `Stream` and inductive program `ListNat`; and no explicit annotation was needed to handle this. These two programs were *well-founded*, however, not all programs will give rise to finite coinductive trees. This leads us to the following definition of well-foundness of logic programs.

DEFINITION 4.12
A logic program P is well-founded if, for any goal G, $P \cup \{G\}$ generates the coinductive tree of finite size.

There are logic programs that allow infinite coinductive trees.

EXAMPLE 4.13
Consider $R(x) \leftarrow R(f(x))$. The coinductive tree arising from this program contains a chain of alternating $\bullet$'s and atoms $R(x)$, $R(f(x))$, $R(f(f(x)))$, etc., yielding an infinite coinductive tree. This tree is a subtree of $\bar{p}(R(x)) \in \&\vee_c\text{-}Tree_{At(1)}$.

In line with the existing practice of functional languages, we want the notion of well-foundness to be transformed into programming practices. For this, a set of syntactic *guardedness* conditions needs to be introduced, compare e.g. with [5, 10, 15]. Coinductive trees we introduced in the previous section allow us to formulate similar guardedness conditions. They correspond to the method of **guarding** (co)recursive function applications **by constructors** in [10, 15]. In our running examples, function symbols `0`, `1`, `s`, `cons`, `scons`, `f` play the role of guarding constructors.

Guardedness check 1 (Presence of Constructors): *If a clause has the form* $P(\bar{t}) \leftarrow [atoms], P(\bar{t'}), [atoms]$, *where* P *is a predicate,* $\bar{t}$, $\bar{t'}$ *are lists of terms, and* [*atoms*] *are*

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

finite (possibly empty) lists of first-order atoms, then at least one term $t_i \in \bar{t}$ must contain a function symbol f.

For example, `Stream` is guarded. But Check-1 is not sufficient to guarantee well-foundness of coinductive trees. Consider the following examples.

EXAMPLE 4.14
Consider the variant of Example 4.13 given by $R(f(x)) \leftarrow R(f(f(x)))$. It generates an infinite coinductive tree, given by a chain of alternating $\bullet$'s and atoms $R(f(x))$, $R(f(f(x)))$, etc.

EXAMPLE 4.15 (`Stream2`)
Another non-well-founded program that satisfies Guardedness check 1 is given below:

```
stream2(scons(x,y)) ← bit(x), stream2(scons(x,y))
```

To address such problems, a second guarding condition is needed.

Guardedness check 2 (Constructor Reduction): *If a clause has the form $P(\bar{t}) \leftarrow [body]$, where P is an n-ary predicate, $\bar{t}$ is a list of terms $t_1, \ldots, t_n$, and $[body]$ is a finite non-empty list of first-order atoms, then, for each occurrence of $P(\bar{t'})$ (with some $\bar{t'} = t'_1, \ldots, t'_n$) in $[body]$, the following conditions must be satisfied. There should exist a term $t_i \in \bar{t}$ such that, there is a function f that occurs in t_i m times ($m \geq 1$) and occurs in t'_i k times with $k < m$. Moreover, if $f \in t_i$ has arguments containing variables $\bar{x_i}$, then $f \in t'_i$ must have arguments containing variables $\bar{x'_i}$, with $\bar{x'_i} \subseteq \bar{x_i}$; if f occurs in t_i but not in t'_i, then all variables $\bar{x'_i} \in t'_i$ must satisfy $\bar{x'_i} \subseteq \bar{x_i}$.*

EXAMPLE 4.16
Suppose we want to define a program that computes and infinite stream of natural numbers: $0, 1, 2, 3, 4, 5, \ldots$ The corresponding logic program will be given by:

```
nats(scons(x,scons(s(x),z))) ← nat(x),nats(scons(s(x),z))
```

It is a well-founded and guarded program, so will result in potentially infinite coinductive derivations featuring coinductive trees of finite size. This program will satisfy Guardedness checks 1 and 2: the function symbol (constructor) `scons` reduces in the body.

EXAMPLE 4.17
In Example 4.14, function symbol f appears twice in the body, while appearing only once in the head; this fails the guardedness check 2.

Note that Guardedness check 2 imposes strict discipline on argument positions at which constructors reduce, and on variables appearing as arguments to the constructors. The next example explains why these restrictions matter.

EXAMPLE 4.18
Consider the following clause: `Q(s(x),y) ← Q(y,y)`
The constructor `s` clearly reduces, and the clause could pass the guardedness check if it was checking only the constructor reduction. However, the goal `Q(s(x),s(x))` would result in an infinite coinductive tree. The problem here is the new variable `y` in the body, in the same argument position as `s(x)`: it allows to form the goals like `Q(s(x),s(x))` falling into infinite loops. To avoid

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

such cases, Guardedness check 2 imposes the restriction on the argument positions and variables. Therefore, the programmer would be forced to change the clause to `Q(s(x),y)` ← `Q(x,y)` to pass the guardedness checks.

Finally, the (co-)recursive nature of the predicates may show only via several clauses in the program. Consider the following example.

EXAMPLE 4.19
Consider programs `P1` and `P2` below. For both programs, Guardedness conditions 1 and 2 are satisfied for every single clause, but the programs give rise to infinite coinductive trees.

P1:
```
Q(cons(x,y)) ← Q2(cons(z,cons(x,y)))
Q2(cons(z,cons(x,y)) ← Q(cons(x,y))
```

P2:
```
Q(cons(x,y)) ← Q2(cons(z,cons(x,y)))
Q2(y) ← Q(y)
```

To address the problem above, a further guardedness check needs to be introduced.

DEFINITION 4.20
Given a logic program P, a goal G, and the coinductive tree for $P \cup \{G\}$, we say T *contains a loop* if there exists a coinductive subtree C of T, such that:
there exists a predicate $Q \in P$ such that $Q(\bar{t})$ appears as an and-node in C, and also $Q(\bar{t'})$ appears as a child and-node of that node in C, for some $\bar{t}$ and $\bar{t'}$.

In this case, we say that atom $Q(\bar{t})$ is a head loop factor, and $Q(\bar{t'})$ is a tail loop factor.

Guardedness check 3 (Detection of Non-guarded Loops): *If a program P satisfies guardedness conditions 1 and 2, do the following. For every clause $C \in P$, such that C has the shape $A \leftarrow B_1, \ldots B_n$, construct the coinductive tree T for A, imposing the following termination conditions during the tree construction:*

i. If T contains a loop with the head and tail factors $Q(\bar{t})$ and $Q(\bar{t'})$, apply Guardedness checks 1 and 2 to $Q(\bar{t}) \leftarrow Q(\bar{t'})$. If the Guardedness checks 1 and 2 are violated for $Q(\bar{t}) \leftarrow Q(\bar{t'})$, terminate the coinductive tree construction for A; report non-guardedness.
ii. If construction of T reaches the leaf nodes and none of the guardedness conditions *(i.)* and *(ii.)* is violated, the program P is guarded.

PROPOSITION 4.21
Guardedness check 3 terminates, for any logic program.

PROOF. Note that a given program P has a finite and fixed number of clauses. If there are n clauses in the given program, only n coinductive trees will be constructed. It remains to show that each tree construction will be terminated in finite time. Given that P contains a finite number of predicates, an infinite coinductive tree T for P would need to contain a loop. If all loops occurring in T are guarded, they could not have constructor reduction infinite number of times, so there should be at least one non-guarded loop. But then the tree construction will be terminated, by item *i*. ■

Note that, although the procedure above requires some computations to be performed, the guardedness checks can be done statically, prior to the program run.

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

EXAMPLE 4.22
Consider the program `P3`:

```
Q(cons(x,y)) ← Q2(cons(x,y))
Q2(cons(x,y)) ← Q(y)
```

It satisfies guardedness checks **1**, **2** and **3**. In particular, coinductive trees for both of its clauses are finite, and show constructor reduction.

The Guardedness checks 1–3 are necessary, but not sufficient conditions for guaranteeing well-foundness of all logic programs. This is why, we include some further checks, involving applying checks 1–3 to program heads modulo some chosen substitutions. We will not go into further details here, but will illustrate the issue by the following example.

EXAMPLE 4.23
Consider the logic program `P4`:

```
Q(s(x),y) ← P(x,y)
P(t(x),y) ← Q(y,y)
```

Each clause passes the Guardedness checks 1-2 trivially, as they do not have immediate loops. Coinductive trees constructed by Check 3 do not exhibit the loops, either, due to the restrictive nature of the term matching. However, for the goal `Q(s(t(x)),s(t(x)))`, the program will give rise to an infinite coinductive tree.

Guardedness conditions of CoALP guarantee that, if a program *P* passed the guardedness checks, then any goal will give rise to only finite coinductive trees. Very often, in functional programming, the guardedness conditions reject some well-founded programs [5, 10, 15]. Termination of recursive programs is in general undecidable, and syntactic guardedness conditions are used only to approximate the notion of termination.

Here, as well as in functional programming, there will be examples of well-founded but non-guarded programs:

EXAMPLE 4.24
The Program `P5` is well-founded but not guarded:

```
Q(s(x),y) ← Q(y,x)
Q(x,s(y)) ← Q(y,x)
```

Furthermore, the guardedness checks are too restrictive to capture the notion of termination in sequential logic programs as given by e.g. SLD-resolution.

EXAMPLE 4.25
The following program is non-well-founded and not guarded in CoALP setting, but terminates if SLD-resolution is used:

```
Q(a) ←
Q(x) ← Q(a)
```

As we discuss in the next section, the program *GC* gives a similar effect.

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

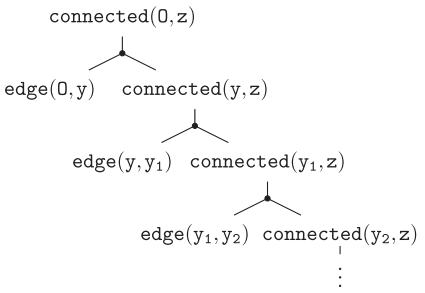

FIGURE 7. The infinite coinductive tree for the programs GC from Example 3.19, GC' from Example 4.27 and GC* from Example 4.26.

Our approach allows us to guard (co-)recursion implicitly, without annotating the predicates as inductive or coinductive, as it was the case in [18, 45]. The advantages of this implicit method of handling (co-)recursive computational resources can be summarized as follows. It solves both difficulties that explicit coinductive resource management causes: in response to ★, the method uniformly treats inductive and coinductive definitions, and it can be used to detect non-well-founded cases in both; and in response to ★★, it is a corecursive process in spirit. Thus, instead of relying on guarantees of loop regularity, it relies on well-foundness of every coinductive tree in the process of lazy infinite derivations.

4.3 Programming with guarded corecursion

We proceed with a case study of how guardedness conditions can be used in logic programming practice.

First, we consider the effects of coalgebraic logic programming on corecursive resource handling by comparing GC (cf. Example 3.19) with `Stream`. GC uses recursion to traverse all the connected nodes in a graph. Two kinds of infinite *SLD*-derivations are possible: computing finite or infinite objects.

EXAMPLE 4.26 (GC*)
Consider the program GC*. Adding the following clause to GC makes the graph cyclic:

$$\texttt{edge(s(s(0)),0)} \leftarrow$$

Taking a query $\leftarrow$ `connected(0,y)` as a goal may lead to an infinite *SLD*-derivation corresponding to an infinite path starting from `0` in the cycle. It would also give rise to infinite coinductive trees, see Figure 7. However, the object that is described by this program, the cyclic graph with three nodes, is finite.

In the standard practice of logic programming, where the ordering of the clauses is as in GC, the program behaves gracefully, giving finitely computed answers, but potentially infinitely many times. But this balance is fragile. For example, the following program, with different ordering of the clauses

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

and of the atoms in the body, results in non-terminating derivations:

EXAMPLE 4.27 (GC')
Let `GC'` denote the logic program

```
connected(x,y) ← connected(z,y),edge(x,z)
connected(x,x) ←
```

together with the database of Example 3.19, *SLD*-derivation loops as follows:

$$\texttt{connected(0,}y\texttt{)} \rightarrow$$
$$\texttt{(connected(}z\texttt{,}y\texttt{),edge(0,}z\texttt{))} \rightarrow$$
$$\texttt{(connected(}z_1\texttt{,}y\texttt{),edge(}z\texttt{,}z_1\texttt{),edge(0,}z\texttt{))} \rightarrow \ldots$$

It never produces an answer as it falls into an infinite loop irrespective of the particular graph in question.

There is a one-step non-deterministic derivation for `connected(0,y)` given by unifying y with 0 (see Definition 3.27.) But there is no coinductive derivation that does that: see Figure 7.

Spelling out non-deterministic semantics (Theorem 3.29),

$$T_1 = \mathit{connected}(0, y);$$
$$T_0 = \mathit{connected}(0,0) \rightarrow \Box.$$

In traditional logic programming, the burden of deciding which programs might result in loops like the one above falls completely to the programmer: semantically, `GC` and `GC'` are equivalent. Moreover, in the Co-LP [18, 45] setting, if the atoms in the programs above are labelled as inductive, the behaviour of Co-LP is exactly as it is for *SLD*-resolution. If, on the contrary, the atoms are marked as coinductive, we may find the derivation loop terminated as 'successful' when we should be warned of its being non-well-founded.

In contrast, compare the coalgebraic semantics of `GC`, `GC'`, `GC*` and `Stream`. Figures 7 and 6 show the difference between the coinductive trees for ill-founded `GC`, `GC'` and `GC*` and well-founded programs like `Stream`. Notably, coinductive definition of `Stream` is well-founded, while traditional inductive definition of `GC*` is not. `GC`, `GC'` and `GC*` give rise to infinite coinductive trees, whereas `Stream` gives rise only to finite coinductive trees.

In CoALP, a set of syntactic guardedness checks 1-3 is embedded, to make sure that only programs that satisfy the semantic notion of well-foundness are allowed in CoALP. Programs like GC, GC' and GC* will be automatically rejected by CoALP's guardedness checks, see Section 6. To make the programs like `GC` guarded, The user will have to reformulate it as follows:

EXAMPLE 4.28 (GC^g)
The program $\texttt{GC}^{\texttt{g}}$ below addresses both non-terminating problem for SLD-derivations for `GC'`, and non-well-foundness of `GC` and `GC*`.

```
connected(x,cons(y,z)) ← edge(x,y),connected(y,z)
      connected(x,nil) ←
             edge(0,0) ←
         edge(x,s(x)) ←
```

The coinductive derivation for it is shown in Figure 8, duly featuring coinductive trees of finite size.

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

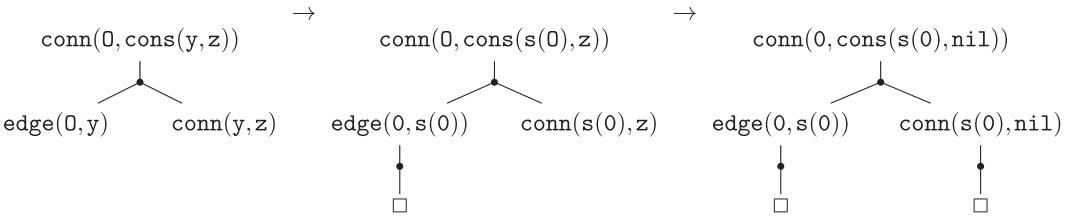

FIGURE 8. A finite and well-founded coinductive derivation for a guarded variant of GC^g; we use `conn` to abbreviate `connected`.

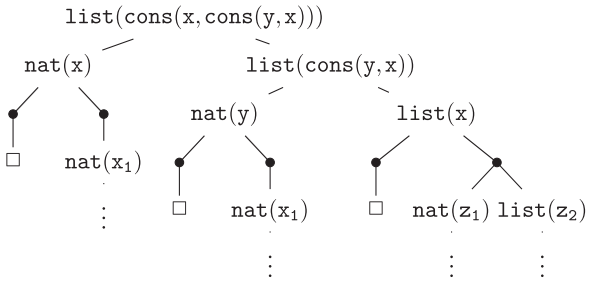

FIGURE 9. An unsound refutation by an and-or parallel derivation tree, with $\theta = \{x/0, y/0, x/nil\}$.

5 Guarding parallelism by guarded corecursion

One of the distinguishing features of logic programming languages is that they allow implicit parallel execution of programs. In the last two decades, an astonishing variety of parallel logic programming implementations have been proposed, see [19] for a detailed survey. The three main types of parallelism used in implementations of logic programs are *and-parallelism*, *or-parallelism* and their combination; see also Section 2.2. The coalgebraic models we discuss in this article exhibit a synthetic form of parallelism: and-or parallelism. The most common way to express and-or parallelism in logic programs is and-or parallel derivation trees [17, 19], see Definition 2.16.

In the ground case, coinductive trees and and-or parallel derivation trees agree, as illustrated by Example 3.8. But as we have discussed, that does not extend. In the general case, in the absence of synchronization, parallel and-or-trees may lead to unsound results.

EXAMPLE 5.1
Figure 9 depicts an and-or parallel derivation tree that finds a refutation $\theta = \{x/0, y/0, x/nil\}$ for the goal `list(cons(x,cons(y,x)))`, although this answer is not sound.

A solution proposed in [17] was given by *composition (and-or parallel derivation) trees*. Construction of composition trees involves additional algorithms that synchronize substitutions in the branches of and-or parallel derivation trees. Composition trees contain a special kind of composition nodes, used whenever both and- and or-parallel computations are possible for one goal. A composition node is a list of atoms in the goal. If, in a goal $G = \leftarrow B_1, \ldots B_n$, an atom B_i is unifiable with $k > 1$ clauses, then the algorithm adds k children (composition nodes) to the node G; similarly for every atom in G that is unifiable with more than one clause. Every such composition node has the form $B_1, \ldots B_n$ and has n and-parallel edges emanating from it. Thus, all possible combinations of or-choices at every and-parallel step are given.

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

Predominantly, the existing parallel implementations of logic programming follow Kowalski's principle [33]:

$$\textit{Programs} = \textit{Logic} + \textit{Control}.$$

This principle separates the control component (backtracking, occur check, goal ordering/selection, parallelization, variable synchronization) from the logical specification of a problem (first-order Horn logic, *SLD*-resolution, unification). Thus the control of program execution becomes independent of programming semantics.

With many parallel solutions on offer [19], some form of resource handling and process scheduling are inevitable ingredients of parallel logic programming as the algorithms of unification and *SLD*-resolution are P-complete [24, 48] and cannot be parallelized in general, see Example 5.1. Parallel implementations of PROLOG typically hide all additional control-handling algorithms at the level of implementation, away from program specification or semantics [19]. The algorithms used for variable synchronization pose a sequential barrier for parallelization.

Several properties are shared by many parallel implementations of PROLOG:

- ⋆ Although and-or-parallelism is called 'implicit parallelism' in the literature [19], it boils down to explicit resource handling at compiler level: this includes both annotating the syntax and maintaining special schedulers/arrays/hash tables to synchronize variable substitutions computed by different processes; these are separated from the language and semantics.
- ⋆⋆ Issues of logic and control are separated to the point that parallel PROLOG systems are usually built as speed-ups to *SLD*-resolution and have neither 'logic' algorithms nor semantics of their own. For composition trees, they are implemented by adding extra features to *SLD*-resolution. Specifically, composition nodes are handled by binding arrays at compiler level.

In the previous sections, we have proposed coinductive trees (cf. Definition 4.1), as an alternative to composition trees. Coinductive trees serve as computational units in lazy (co)recursive derivations, and therefore, these coinductive tree transitions can be parallelized, as well. For guarded logic programs, coinductive derivations allow for parallel and even non-deterministic implementations, as Sections 4.1 and 6 explain. Here, we explain the two levels of parallelism in CoALP:

Level 1: Parallel construction of coinductive trees.

Comparing coinductive derivation trees with and-or parallel derivation trees, coinductive trees are more intrinsic: and-or parallel trees have mgu's built into a single tree, whereas mgu's are restricted to term-matching within the coinductive tree. Taking issues of variable substitution from the level of individual leaves to the level of trees affects computations at least in two ways. Parallel proof-search in branches of a coinductive tree does not require synchronization of variables in different branches: they remain synchronized *by construction* of the coinductive tree. We illustrate with `ListNat`.

EXAMPLE 5.2

The coinductive trees of Figure 11 agree with the first part of the and-or parallel derivation tree for `list(cons(x,cons(y,x)))` in Figure 9. But the top left coinductive tree has leaves `nat(x)`, `nat(y)` and `list(x)`, whereas the and-or parallel derivation tree follows those nodes, using substitutions determined by mgu's. Moreover, those substitutions need not be consistent with each other: not only are there two ways to unify each of `nat(x)`, `nat(y)` and `list(x)`, but also there is no consistent substitution for `x` at all. In contrast, the coinductive trees handle such cases lazily.

Term-matching in coinductive trees permits the construction of every branch in a coinductive tree independently of the other branches. Moreover, for programs that are guarded by constructors,

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

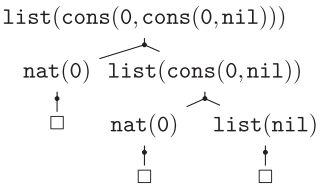

FIGURE 10. An and-or parallel derivation for the goal `list(cons(O,cons(O,nil)))`.

such as `Stream` and `ListNat`, we avoid infinite branches or an infinite number of variables in a single tree. Since both term-matching and guardedness are components of the 'logic' algorithm of coinductive derivation, the Kowalski's principle can be reformulated for CoALP as follows:

CoALP = Logic ***is*** *Control*.

This distinguishes two approaches:

Parallel LP = and-or parallel derivation trees + explicit handling of parallel resources at compiler level; and

CoALP = coinductive derivation trees + implicit handling of parallel resources 'by program construction'.

We start by illustrating ground cases of parallel derivations: these can be parallelized straightforwardly, and coinductive trees and and-or parallel derivation trees coincide. We consider the inductive program `ListNat`, although a similar case-study could be done with a coinductive logic program such as `Stream`.

EXAMPLE 5.3
Consider the and-or parallel derivation tree for `ListNat` with goal `list(cons(0,cons(0,nil)))` in Figure 10.

No additional syntactic annotations or variable synchronization algorithms is required by CoALP when extending from ground cases to the full fragment of first-order Horn logic with recursion and corecursion. Not only termination, but also soundness of parallelism will be guarded by program construction; see also [32].

EXAMPLE 5.4
Consider the coinductive derivation for the goal `list(cons(x,cons(y,x)))` given in Figure 11. In contrast to the and-or parallel derivation tree, and owing to the restriction of unification to term matching, every coinductive tree in the derivation pursues fewer variable substitutions than the corresponding and-or parallel derivation tree does, cf. Figure 9. This allows one to keep variables synchronized while pursuing parallel proof branches in the tree. In particular, coinductive derivation of Figure 11 will report failure, as required for this example.

Level 2: Parallel transitions between coinductive trees.

Consider the leftmost coinductive tree of Figure 11. It has three leaves with two distinct variables. Hence, three independent mgu's can be computed to unfold that tree; and the three tree transitions can be done in parallel. As the lazy nature of coinductive trees and guardedness checks of CoALP insure both soundness and termination of computations at the level of each individual tree, this opens

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

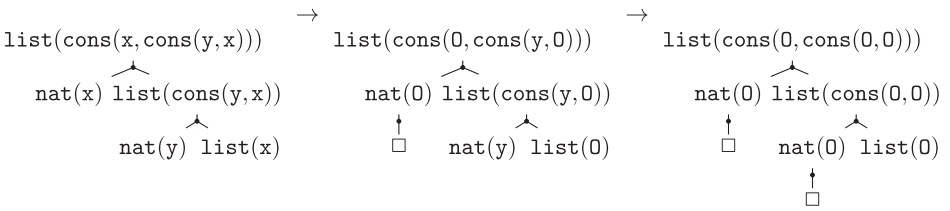

FIGURE 11. A coinductive derivation for the goal `list(cons(x,cons(y,x)))`.

a possibility for parallel proof search through the state space of such trees. We discuss this in detail in Section 6.

To conclude, CoALP gives a different view of parallel resource handling:

(1) We avoid explicit resource handling either at 'logic' or 'control' level; instead, we use implicit methods to control parallel resources.
In particular, we restrict unification to term matching: in contrast to the inherently sequential unification algorithm [11], it is parallelizable. As a result, parallel proof search in separate branches of a coinductive tree does not require explicit synchronization of variables.
Static guardedness checks of CoALP, introduced to guard corecursion, in fact insure that parallel scheduling of computations within the coinductive trees will never fall into a non-terminating thread; and parallel scheduling of coinductive tree transitions will never produce unsound results. Again, this is achieved without introducing new syntax, just by the guarded program construction.

(2) The issues of logic and control are now bound together: coinductive trees provide both logic specification and resource control. Moreover, CoALP comes with its own coalgebraic semantics that accounts for observational behaviour of coinductive derivations.

As the next section explains, this approach to parallelism can be viable and efficient. See also [32] for a detailed study and testing of CoALP's parallel features in ground, Datalog, and full first-order case.

6 Implementation

In [31], we developed the first minimal prototype of CoALP in PROLOG, to show the feasibility of the coalgebraic logic programming approach, see *CoALP Prototype-1* in [44]. However, it did not make use of parallelization in modern computer architectures and was constrained by the mechanisms employed by the underlying PROLOG engine. Here, we present a new binary standalone implementation engineered using the **Go** programming language [47]; available as *CoALP Prototype-2* in [44]. Its most important new feature is the use of **Go**'s built in support for multithreading to achieve parallelization by using *goroutines* which are coroutines that can be executed in distinct threads. This new implementation also features two levels of parallelism (for coinductive trees and their transitions), static guardedness checks, and implicit handling of corecursion and parallelism. In this section, we describe the most important features arising in the implementation of CoALP.

Construction of Coinductive trees (cf. Definition 4.1) lies at the heart of CoALP's implementation. They are implemented by linking structural records (structs) which represent

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

or-nodes and and-nodes through the use of arrays and pointers. And-nodes represent goal terms and contain a list of pointers to clauses that have heads which are still unifiable with the goal. An and-node with a list containing at least one such pointer is regarded as an open node. The root of any coinductive tree is an and-node constructed by the initial goal.

Guardedness plays an important role in CoALP implementation, as Sections 4 and 5 explain. For the proper operation of the CoALP algorithm, it needs to be ensured that a derivation step never produces an infinite and therefore non-well-founded coinductive tree. This would block the search process by taking up infinite time to expand the tree. We have incorporated the *Guardedness checks* in CoALP (cf. Section 4.2); they are used to statically check the input programs, prior to the program run. Note that, in line with lazy corecursion in functional languages, while a coinductive tree may only be finite, the coinductive derivation may still be infinite (cf. `Stream` in Figure 6).

Coinductive derivations are transitions of coinductive trees. Whether the CoALP implementation is viewed as a sequential or parallel process, it can be described as follows. Construction of coinductive derivations for a given input program and goal is modelled as a uniform cost search through the graph of coinductive trees connected by the derivation operation. A derivation step here is constrained to first order unification of the first unifiable open node that has the lowest level in the tree; cf. Definition 3.26 and Figures 6 and 11. Other strategies, including non-deterministic methods are possible for selecting such open nodes; thereby determining substitutions for new coinductive tree transitions. Only a very thin layer of sequential control in the implementation for this search is needed in the form of a priority search queue.

EXAMPLE 6.1
Looking at the `ListNat` program from Example 2.6, the tree with root `list(cons(x, cons(y, x)))` is connected to `list(cons(0, cons(y, 0)))` by unification of the open node `nat(x)` with `nat(0)`. This step is also shown in Figure 4. The following derivation and the resulting coinductive tree for `list(cons(0, cons(0, 0)))` contains no unifiable open nodes—note that `list(0)` cannot be unified with any clause head of the input program.

Using the substitution length of all the substitutions in the derivation chain as priority ranking, we gain an enumeration order even for a potentially infinite lazy derivation processes. Therefore, while an infinite number of coinductive trees can in principle be produced for the goal `list(x)`, the algorithm returns `list(nil)`, `list(cons(0,nil))` and then `list(cons(s(0),nil))` in a finite number of time-steps and keeps producing finite coinductive trees thereafter. Running CoALP [44] for `list(x)`, we get as output the substitutions for the first three success trees:

1 $\{x/nil\}$,
3 $\{x/cons(x_1,y_1),x_1/0,y_1/nil\}$ and
4 $\{x/cons(x_1,y_1),x_1/s(x_2),x_2/0,y_1/nil\}$.

Each possible coinductive tree will be produced after finite time, but since there may be infinitely many such trees, the coinductive derivations are implemented as lazy corecursive computations. Contrast this to PROLOG which produces the solutions `list(nil)` , `list(cons(0,nil))` , `list(cons(0,cons(0,nil)))`, … but never `list(cons(s(0),nil))` for the `ListNat` program and goal `list(x)`. Thereby, it does not generate the same set of solutions even if run indefinitely and does not discover some of the solutions that CoALP does.

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

The CoALP implementation allows for various forms of output for a query. Besides reporting solutions as for the `ListNat` program above it can also show just the root nodes or full trees serialized in form of terms for generated trees that match user specified properties.

EXAMPLE 6.2
When queried for the root nodes of trees with new substitutions to the root node for the query `stream(X)` in the `Stream` program with a maximum number of 2 substitutions the CoALP implementation prints:

```
stream(scons(V1,V2...))
stream(scons(0,V3...))
stream(scons(1,V4...))
stream(scons(1,scons(V5,V6...)))
stream(scons(1,scons(0,V7...)))
stream(scons(1,scons(1,V8...)))
stream(scons(0,scons(V9,V10...)))
stream(scons(0,scons(0,V11...)))
stream(scons(0,scons(1,V12...)))
```

Variable names have been shortened for brevity. The … denote variables which are involved in the expansion of the tree by coinductive predicates. When the user-supplied limit of maximum substitutions or solutions is reached, he will be asked if the implementation should continue up to a new limit if further derivations are possible; see [44] for more details.

A new approach to Backtracking is taken, as CoALP explores simultaneously several and-or-choices in a coinductive tree. In contrast to PROLOG, no trail stack is maintained and no backtracking (in the classical sense of [35]) is needed. If a coinductive tree has no open unifiable nodes, it will simply be discarded. If alternative mgu's existed during the derivation steps, they open up different branches in coinductive derivations. Therefore, CoALP implicitly represents alternative mgu's by coinductive trees in the priority search queue. The only time variable bindings may be undone is when checking for unifiability of terms during the derivation step. However, this is only done on copies of the original terms to ensure thread safety and to avoid unnecessary locks and therefore sequential barriers. Furthermore, this is done locally and does not characterize or regulate the overall global search flow.

Parallelization of coinductive trees. Given that no infinite derivation tree can be generated by a guarded program, the CoALP approach provides multiple points where parallelization takes place, while still enumerating every possible coinductive success subtree. The use of term matching to traverse and expand trees allows for parallelization of work without explicit variable synchronization while operating directly on a single tree.

However, if the coinductive trees are small or few open nodes exist, such as in the running examples `Stream` and `ListNat`, the setup and initial communication overhead between parallel threads that process the tree does not usually offset speedup that can be achieved. Therefore, it is dynamically decided during execution whether a program generates sufficiently complex coinductive trees to warrant this parallelization strategy. Future research will focus on efficient heuristics to decide how this trade-off should be made.

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

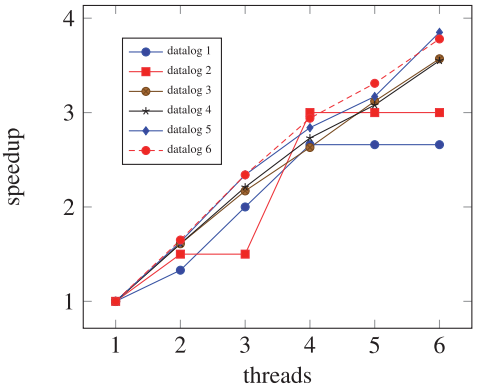

FIGURE 12. Speedup of Datalog programs, relative to the base case with 1 thread, with different number of threads expanding the derivation tree.

Term matching can be performed in parallel, but if the terms are small, no practical speedup will be obtained when working with multiple threads. In such cases, it is more efficient to perform distinct term matching operations in parallel by dispatching work on multiple coinductive trees in parallel.

Ground logic programs do not need transitions between the coinductive trees to complete the computation. Logic programs containing variables but no function symbols of arity $n > 0$ can all be soundly translated into finitely presented ground logic programs. The most famous example of such a language is Datalog [24, 48]. The advantages of Datalog are easier implementations and a greater capacity for parallelization.

Figure 12 shows the speedup that can be gained by constructing and-or parallel trees for Datalog programs in our system. The Datalog programs are randomly generated and can be examined in [44]. As can be seen in Figure 12, the speedup is significant and scales with the number of threads; see also [32].

Parallelization of coinductive derivations is more efficient than parallelization within one coinductive tree for programs like `ListNat` and `Stream`. On the search queue level of the algorithm, multiple trees that still have open nodes and possible derivations are dispatched to one or more worker threads. They perform the coinductive derivation steps in parallel. To keep communication minimal, the coinductive trees are compacted by e.g. pruning closed leaves and shortening chains that have no branches in the tree. Since expanding and checking coinductive trees does not always take the same amount of time for each tree, some worker threads might return results earlier than others and thereby disrupt the enumeration order. So, we do not allow them to show results immediately and directly to the user. CoALP guarantees that success trees which are enumerated sequentially will also be found when working in multithreaded context albeit maybe later. Returning results in the enumeration order of substitution lengths to the user can still be achieved by a little more sequential overhead. For example, the user can specify the option to buffer and sort success coinductive trees until it is guaranteed that no lower order coinductive trees are still being processed or are in the priority search queue.

Considering the other direction of reducing sequential overhead in maintaining the search queue, there is the possibility of using complementary enumeration schemes and thereby partition the search queue into smaller queues that each worker thread maintains on its own. However, this may shift the

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

order of solutions since some worker threads may enumerate only solutions that are computationally easier to find. Thereby a trade-off is to be made between maintaining a perfect ordering or faster processing of coinductive trees. At any rate, the derivations remain sound by the program guardedness and coinductive tree construction, cf Sections 4 and 5; and this allows for a range of experiments on parallelization for the future.

7 Conclusions and future work

The main feature of the coalgebraic logic programming approach is its generality: it is suitable for both inductive and coinductive logic programs, for programs with variable dependencies or not, and for programs that are unification-parallelizable or inherently sequential. Many distinctions that led to a variety of engineering solutions in the design of corecursive and parallel logic programs [18, 19, 45] are erased here, with resource-handling delegated to a logic algorithm; and issues of logic and control, semantics and execution, become inseparable.

The original contributions of this article relative to the earlier papers [27, 29, 30] are the Coalgebraic calculus of infinite trees (Section 3.1), operational semantics for non-deterministic derivations (Section 3.4), extended Guardedness conditions for CoALP (Section 4.2), and Parallel and Corecursive Implementation of CoALP in Go (Section 6). Additionally, the study develops a unifying theory and notation for parallelism and corecursion in logic programming, putting a new perspective on earlier results [27, 29, 30]. Proofs of Soundness and Completeness Theorems 3.21, 3.29, 4.10 appear here for the first time.

The current work is focused on refining CoALP's guardedness checks and termination conditions for derivations in inductive and coinductive cases. In future, we plan to investigate the integration of coalgebraic logic programming with methods of resource handling in state-of-the-art coinductive logic programming [18, 19, 45], as well as in modern parallel logic programming systems [19]. Furthermore, we would like to investigate whether coalgebraic logic programming has potential to play a positive role in type inference, cf. [3]. The work is on the way to implement CoALP in Haskell, to allow easier integration into Haskell, Agda, Hume, Idris or Epigram type inference.

The analysis of this study can be extended to more expressive logic programming languages, such as [16, 21, 38, 42], also to functional programming languages in the style of [3, 40]. We deliberately chose our running examples to correspond to definitions of inductive or coinductive types in such languages.

The key fact driving our analysis has been the observation that the implication $\leftarrow$ acts at a meta-level, like a sequent rather than a logical connective. That observation extends to first-order fragments of linear logic and the Logic of Bunched Implications [16, 42]. So we plan to extend the work in the study to logic programming languages based on such logics.

The situation regarding higher-order logic programming languages such as λ-*PROLOG* [38] is more subtle. Despite their higher-order nature, such logic programming languages typically make fundamental use of sequents. So it may well be fruitful to consider modelling them in terms of coalgebra too, albeit probably on a sophisticated base category such as a category of Heyting algebras.

Acknowledgments

The work was supported by the Engineering and Physical Sciences Research Council, UK: Postdoctoral Fellow in TCS grant EP/F044046/1-2, EPSRC First Grant EP/J014222/1 and EPSRC Grant EP/K031864/1 (to E.K.). The work was supported by Royal Society grant 'Universal Algebra

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

and its dual: monads and comonads, Lawvere theories and what?', EPSRC grant EP/K028243/1, and SICSA Distinguished Visiting Fellow grant (to J.P.).

References

[1] K. Ali and R. Karlsson. Full prolog and scheduling or-parallelism in muse. *International Journal Of Parallel Programming*, **19**, 445–475, 1991.
[2] G. Amato, J. Lipton, and R. McGrail. On the algebraic structure of declarative programming languages. *Theor. Comput. Sci.*, **410**, 4626–4671, 2009.
[3] D. Ancona, G. Lagorio, and E. Zucca. Type inference by coinductive logic programming. In *TYPES*, vol. 5497 of *LNCS*, pp. 1–18, 2009.
[4] M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice-Hall, 1990.
[5] Y. Bertot and E. Komendantskaya. Inductive and coinductive components of corecursive functions in Coq. *ENTSC*, **203**, 25–47, 2008.
[6] F. Bonchi and U. Montanari. Reactive systems, (semi-)saturated semantics and coalgebras on presheaves. *Theor. Comput. Sci.*, **410**, 4044–4066, 2009.
[7] F. Bonchi and F. Zanasi. Saturated semantics for coalgebraic logic programming. In *CALCO*, vol. 8089 of *Lecture Notes in Computer Science*, pp. 80–94. Springer, 2013.
[8] R. Bruni, U. Montanari, and F. Rossi. An interactive semantics of logic programming. *TPLP*, **1**, 647–690, 2001.
[9] M. Comini, G. Levi, and M. C. Meo. A theory of observables for logic programs. *Inf. Comput.*, **169**, 23–80, 2001.
[10] T. Coquand. Infinite objects in type theory. In *Types for Proofs and Programs, Int. Workshop TYPES'93*, vol. 806 of *LNCS*, pp. 62–78. Springer-Verlag, 1994.
[11] C. Dwork, P. Kanellakis, and J. Mitchell. On the sequential nature of unification. *J. Logic Prog.*, **1**, 35–50, 1984.
[12] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A model-theoretic reconstruction of the operational semantics of logic programs. *Inf. Comput.*, **103**, 86–113, 1993.
[13] M. Falaschi, G. Levi, C. Palamidessi, and M. Martelli. Declarative modeling of the operational behavior of logic languages. *TCS*, **69**, 289–318, 1989.
[14] M. Gabrielli, G. Levi, and M. Meo. Observable behaviors and equivalnences of logic programs. *Information and Computation*, **122**, 1–29, 1995.
[15] E. Giménez. Structural recursive definitions in type theory. In *ICALP*, vol. 1443 of *LNCS*, pp. 397–408. Springer, 1998.
[16] J.-Y. Girard. Linear logic. *Theor. Comput. Sci.*, **50**, 1–102, 1987.
[17] G. Gupta and V. Costa. Optimal implementation of and-or parallel prolog. In *PARLE'92*, pp. 71–92, 1994.
[18] G. Gupta et al. Coinductive logic programming and its applications. In *ICLP 2007*, vol. 4670 of *LNCS*, pp. 27–44, 2007.
[19] G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. Hermenegildo. Parallel execution of prolog programs: a survey. *ACM Trans. Computational Logic*, pp. 1–126, 2012.
[20] M. Hermenegildo and K. J. Greene. &-prolog and its performance: exploiting independent and-parallelism. In *ICLP*, pp. 253–268, 1990.
[21] J. S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Inf. Comput.*, **110**, 327–365, 1994.
[22] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of EATCS*, **62**, 222–259, 1997.

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016

[23] M. Jaume. On greatest fixpoint semantics of logic programming. *J. Log. Comput.*, **12**, 321–342, 2002.
[24] P. C. Kanellakis. Logic programming and parallel complexity. In *Foundations of Deductive Databases and Logic Prog.*, pp. 547–585. Morgan Kaufmann, 1988.
[25] G. M. Kelly. Coherence theorems for lax algebras and for distributive laws. In *Category seminar*, vol. 420 of *LNM*, pp. 281–375, 1974.
[26] Y. Kinoshita and J. Power. A fibrational semantics for logic programs. In *Proc. Int. Workshop on Extensions of Logic Programming*, vol. 1050 of *LNAI*, 1996.
[27] E. Komendantskaya, G. McCusker, and J. Power. Coalgebraic semantics for parallel derivation strategies in logic programming. In *AMAST'2010*, vol. 6486 of *LNCS*, 2010.
[28] E. Komendantskaya and J. Power. Fibrational semantics for many-valued logic programs: Grounds for non-groundness. In *JELIA'08*, vol. 5293 of *LNCS*, pp. 258–271, 2008.
[29] E. Komendantskaya and J. Power. Coalgebraic derivations in logic programming. In *CSL*, LIPIcs, pp. 352–366. Schloss Dagstuhl, 2011.
[30] E. Komendantskaya and J. Power. Coalgebraic semantics for derivations in logic programming. In *CALCO, LNCS*, pp. 268–282. Spinger, 2011.
[31] E. Komendantskaya, J. Power, and M. Schmidt. Coalgebraic logic programming: implicit versus explicit resource handling. In *Coinductive Logic Programming Workshop, ICLP'12*, 2012.
[32] E. Komendantskaya, M. Schmidt, and J. Heras. Exploiting parallelism in coalgebraic logic programming. In *Accepted for Wessex Seminar ENTCS Post-Proceedings*, 2013.
[33] R. Kowalski. *Logic for Problem Solving*. Elsevier, 1979.
[34] J. Lambek and P. Scott. *Higher Order Categorical Logic*. Cambridge University Press, 1986.
[35] J. Lloyd. *Foundations of Logic Programming*, 2nd edn. Springer-Verlag, 1987.
[36] E. L. Lusk, D. H. D. Warren, and S. Haridi. The aurora or-parallel prolog system. *New Generation Computing*, **7**, 243–273, 1990.
[37] Z. Majkic. Coalgebraic semantics for logic programming. In *18th Workshop on (Constraint) Logic Programming, WLP 2004, March 04-06*, 2004.
[38] D. Miller and G. Nadathur. Higher-order logic programming. In *ICLP*, pp. 448–462, 1986.
[39] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
[40] L. C. Paulson and A. W. Smith. Logic programming, functional programming, and inductive definitions. In *ELP*, pp. 283–309, 1989.
[41] E. Pontelli and G. Gupta. On the duality between or-parallelism and and-parallelism in logic programming. In *Euro-Par*, pp. 43–54, 1995.
[42] D. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*, vol. 26 of *Applied Logic Series*. Kluwer Academic Publishers, 2002.
[43] J. Rutten. Universal coalgebra: a theory of systems. *TCS*, 2000.
[44] M. Schmidt and E. Komendantskaya. Coalgebraic logic programming (coalp): Implementation. Prototypes 1 and 2, 2012. www.computing.dundee.ac.uk/staff/katya/CoALP/.
[45] L. Simon et al. Co-logic programming: extending logic programming with coinduction. In *ICALP*, vol. 4596 of *LNCS*, pp. 472–483. Springer, 2007.
[46] L. Sterling and E. Shapiro. *The art of Prolog*. MIT Press, 1986.
[47] M. Summerfield. *Programming in Go: Creating Applications for the 21st Century*. Addison-Wesley, 2012.
[48] J. D. Ullman and A. V. Gelder. Parallel complexity of logical query programs. *Algorithmica*, **3**, 5–42, 1988.

Received 26 December 2012

Downloaded from http://logcom.oxfordjournals.org/ at University of Dundee on May 4, 2016